bytes lounge

Java articles, how-to's, examples and tutorials

MENU

# EJB multiple datasource transaction example

08 May 2014
By Gonçalo Marques

ejb java-ee jpa jta xa

In this article we will see how to configure EJB transactions across multiple datasources

## Introduction

Sometimes we may need our EJB transactions to spawn across multiple datasources. For this to be possible one must take advantage of the enterprise container Transaction Manager and the Java transaction API (JTA).

The Transaction Manager will be responsible for coordinating the multiple datasource transaction. This transaction coordination will be transparent for the participating EJBs.

We must make sure that the Resource Managers (drivers) we will be using are **XA** compatible, ie. they are able to participate in distributed transactions.

Each participating datasource must be associated with a XA compatible driver. These datasources will be referenced by a distinct **JTA** Persistence Unit in the application (persistence.xml).

Since the container will be managing the transaction through the Transaction Manager, this means that only Container Managed Transaction (**CMT**) EJBs may transparently participate in this kind of distributed transaction.

This article considers the following environment:

1. Ubuntu 12.04
2. JDK 1.7.0.21
3. Glassfish 4.0

## The Datasources

We will configure a couple of datasources to be used in this article: **dataSource1** and **dataSource2**.

MySQL will be our database. Since we need a couple of datasources, we will create a couple of database users, each one of them associated with a distinct database:

**Configured database users**

**Datasource 1**
**Database:** DATABASE1
**Username:** user1
**Database:** passwd1

**Datasource 2**
**Database:** DATABASE2
**Username:** user2
**Database:** passwd2

In each database we create a testing table that will hold the records created by the application:

**Configured tables**

**Datasource 1**

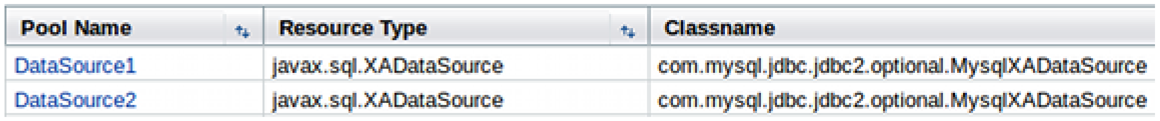
```
CREATE TABLE TABLE_ONE (
TABLE_ONE_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
VALUE VARCHAR(32) NOT NULL
);
```

**Datasource 2**

```
CREATE TABLE TABLE_TWO (
TABLE_TWO_ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
VALUE VARCHAR(32) NOT NULL
);
```

As we have said previously, the datasources must be configured with a **XA** compatible driver. So when we create the corresponding connection pool in Glassfish for each resource, we select the pool Resource Type as **javax.sql.XADataSource**:

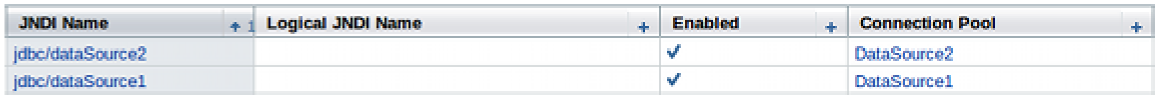
XA Connection Pool configuration



| Pool Name | Resource Type | Classname |
|---|---|---|
| DataSource1 | javax.sql.XADataSource | com.mysql.jdbc.jdbc2.optional.MysqlXADataSource |
| DataSource2 | javax.sql.XADataSource | com.mysql.jdbc.jdbc2.optional.MysqlXADataSource |

Now we configure the corresponding JDBC resources - providing the JNDI name - associating each one with the previously configured connection pools:

JDBC Resource configuration



| JNDI Name | Logical JNDI Name | Enabled | Connection Pool |
|---|---|---|---|
| jdbc/dataSource2 | | ✓ | DataSource2 |
| jdbc/dataSource1 | | ✓ | DataSource1 |

## The Persistence Context

Now we configure a couple of Persistence Units in **persistence.xml**, each one associated with a distinct datasource we configured previously:

/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

  <persistence-unit name="PersistenceUnit1" transaction-type="JTA">
      <jta-data-source>jdbc/dataSource1</jta-data-source>
      <class>com.byteslounge.entity.TableOne</class>
  </persistence-unit>
  <persistence-unit name="PersistenceUnit2" transaction-type="JTA">
      <jta-data-source>jdbc/dataSource2</jta-data-source>
      <class>com.byteslounge.entity.TableTwo</class>
  </persistence-unit>

</persistence>
```

Since the distributed transaction will be managed by the container Transaction Manager, we configure each Persistence Unit transaction type as **JTA**. This also means that the Persistence Context associated with each Persistence Unit will be also managed by the container.

## JPA Entities

Now we define a couple of JPA entities that will represent the tables from each of the databases:

**TableOne.java**

```java
package com.byteslounge.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "TABLE_ONE")
public class TableOne {

  @Id
  @Column(name = "TABLE_ONE_ID", nullable = false)
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int tableOneId;

  @Column(name = "VALUE", nullable = false)
  private String value;

  public int getTableOneId() {
    return tableOneId;
  }

  public void setTableOneId(int tableOneId) {
    this.tableOneId = tableOneId;
  }

  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;
  }


}
```

TableTwo.java

```
package com.byteslounge.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "TABLE_TWO")
public class TableTwo {

  @Id
  @Column(name = "TABLE_TWO_ID", nullable = false)
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int tableTwoId;

  @Column(name = "VALUE", nullable = false)
  private String value;

  public int getTableTwoId() {
    return tableTwoId;
    return tableTwoId;
  }

  public void setTableTwoId(int tableTwoId) {
    this.tableTwoId = tableTwoId;
  }

  public String getValue() {
    return value;
  }

  public void setValue(String value) {
    this.value = value;

  }

}
```

## The EJBs

Finally we define the EJBs that will participate in the distributed transaction.

ServiceOne.java

```
package com.byteslounge.ejb;

import javax.ejb.Local;
import com.byteslounge.entity.TableOne;

@Local
public interface ServiceOne {

  void save(TableOne tableOne);

}
```

ServiceOneBean.java

```
package com.byteslounge.ejb;
```

```
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.byteslounge.entity.TableOne;

@Stateless
public class ServiceOneBean implements ServiceOne {

  @PersistenceContext(unitName = "PersistenceUnit1")
  private EntityManager em;

  @Override

  public void save(TableOne tableOne) {
    em.persist(tableOne);
  }

}
```

The first EJB is quite straight forward: We only defined a single method that will persist an entity of type **TableOne**. Note that we are injecting an Entity Manager that will serve as an interface to interact with **PersistenceUnit1** Persistence Context.

Now we define a second EJB:

**ServiceTwo.java**

```
package com.byteslounge.ejb;

import javax.ejb.Local;
import com.byteslounge.entity.TableTwo;

@Local
public interface ServiceTwo {

  void save(TableTwo tableTwo);

}
```

**ServiceTwoBean.java**

```java
package com.byteslounge.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.byteslounge.entity.TableTwo;

@Stateless
public class ServiceTwoBean implements ServiceTwo {

  @PersistenceContext(unitName = "PersistenceUnit2")
  private EntityManager em;


  @Override
  public void save(TableTwo tableTwo) {
    em.persist(tableTwo);
    throw new RuntimeException("Rollback transaction!");
  }

}
```

Once again we defined a single method that will save an entity of type **TableTwo**. This method is intentionally throwing a **RuntimeException** that will result in the global distributed transaction being rolled back.

In this EJB we are injecting an Entity Manager that is associated with **PersistenceUnit2** Persistence Context.

Finally we define an EJB that will serve as a facade to the distributed transaction:

**TransactionalService.java**

```java
package com.byteslounge.ejb;

import javax.ejb.Local;

import com.byteslounge.entity.TableOne;
import com.byteslounge.entity.TableTwo;

@Local
public interface TransactionalService {
  void save(TableOne tableOne, TableTwo tableTwo);

}
```

TransactionalServiceBean.java

```
package com.byteslounge.ejb;

import javax.ejb.EJB;
import javax.ejb.Stateless;

import com.byteslounge.entity.TableOne;
import com.byteslounge.entity.TableTwo;

@Stateless
public class TransactionalServiceBean implements TransactionalService {

  @EJB
  private ServiceOne serviceOne;

  @EJB
  private ServiceTwo serviceTwo;

  @Override
  public void save(TableOne tableOne, TableTwo tableTwo) {
    serviceOne.save(tableOne);
    serviceTwo.save(tableTwo);
  }

}
```

We defined a method that receives a couple of entities, each one of a distinct type: **TableOne** and **TableTwo**. Each injected EJB is responsible for persisting one of the entities.

The transaction will be propagated to each nested EJB call. If both nested EJB calls succeed, the container will signal the Transaction Manager that the distributed transaction may commit (the commit protocol used by the Transaction Manager and each of the Resource Managers will not be discussed in this article).

## Testing

Let us define a simple servlet in order to test the distributed transaction:

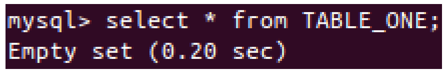
**Testing Servlet**

```
package com.byteslounge.servlet;

import java.io.IOException;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.byteslounge.ejb.TransactionalService;
import com.byteslounge.entity.TableOne;
import com.byteslounge.entity.TableTwo;

@WebServlet(name = "testingServlet", urlPatterns = { "/testing" })
public class TestingServlet extends HttpServlet {

  private static final long serialVersionUID = 1L;

  @EJB
  private TransactionalService transactionalService;

  @Override
  protected void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {

    TableOne tableOne = new TableOne();
    tableOne.setValue("value1");

    TableTwo tableTwo = new TableTwo();
    tableTwo.setValue("value2");

    try {
      transactionalService.save(tableOne, tableTwo);
    } catch (Exception e) {
      e.printStackTrace();
    }

  }

}
```

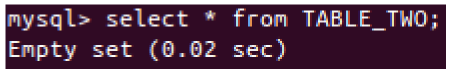
When we access the servlet, the **ServiceTwo** EJB will throw our intentional exception so the distributed transaction will be rollback:

```
javax.ejb.EJBTransactionRolledbackException
...
Caused by: javax.ejb.TransactionRolledbackLocalException: Exception thrown from bean
...
Caused by: java.lang.RuntimeException: Rollback transaction!
```

And both tables in each database will still be empty:



```
mysql> select * from TABLE_ONE;
Empty set (0.20 sec)
```



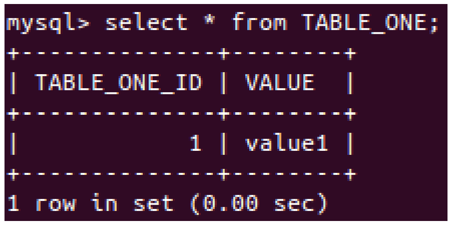
```
mysql> select * from TABLE_TWO;
Empty set (0.02 sec)
```

Now if we modify the **ServiceTwo** EJB method to complete without throwing the runtime exception:

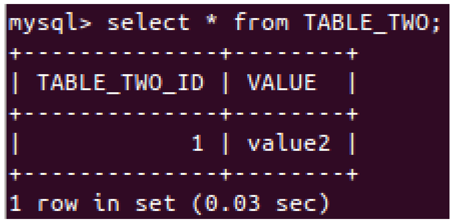
**Modified ServiceTwo EJB method**

```
@Override
public void save(TableTwo tableTwo) {
  em.persist(tableTwo);
  // throw new RuntimeException("Rollback transaction!");
}
```

When we access the servlet, the distributed transaction will complete successfully and the data will be persisted in both tables / databases:



```
mysql> select * from TABLE_ONE;
+--------------+--------+
| TABLE_ONE_ID | VALUE  |
+--------------+--------+
|            1 | value1 |
+--------------+--------+
1 row in set (0.00 sec)
```



```
mysql> select * from TABLE_TWO;
+--------------+--------+
| TABLE_TWO_ID | VALUE  |
+--------------+--------+
|            1 | value2 |
+--------------+--------+
1 row in set (0.03 sec)
```

The article source code is available for download at the end of this page.

## Download source code from this article

**Download link:** ejb-multiple-datasource-transaction-example.zip

## Related Articles

- Java EE EJB transaction propagation (@TransactionAttribute) tutorial
- Java EE EJB concurrency (ConcurrencyManagement, Lock and LockType)
- EJB passivation and activation example
- Java EE Stateful Session Bean (EJB) example

## Comments

Post Comment

**Rashid Mahmood**
04 August 2014

Simplest way of explanation. Keep it up and thanks

Reply

**azzzo**
05 December 2016

Great man

Reply

### About the author

Gonçalo Marques is a Software Engineer with several years of experience in software development and architecture definition. During this period his main focus was delivering software solutions in banking, telecommunications and governmental areas. He created the Bytes Lounge website with one ultimate goal: share his knowledge with the software development community. His main area of expertise is Java and open source.

GitHub profile: https://github.com/gonmarques

He is also the author of the **WiFi File Browser** Android application:

## SOCIAL

Like Follow Share

## ABOUT

Gonçalo Marques

Java and Open Source enthusiast

GitHub profile

Bytes Lounge
582 likes

Like Page Share

Be the first of your friends to like this

## SUBSCRIBE

Receive the latest articles in your mailbox

Subscribe

## POPULAR TAGS

java8 cdi spring jpa java jsf java-ee mvc web

## ARCHIVE

2015 (10)
2014 (32)
2013 (55)
2012 (9)

Java Collections
Java IO
Design Patterns
Java Concurrency
Java Reflection
Java 8

CDI
EJB
JPA
Web
JSF

JAAS

Spring Core
Spring MVC

Android
Maven
Postfix
New Relic