# Red Hat JBoss Enterprise Application Platform 7.2

## Managing Transactions on JBoss EAP

For Use with Red Hat JBoss Enterprise Application Platform 7.2

Last Updated: 2019-09-26

# Red Hat JBoss Enterprise Application Platform 7.2 Managing Transactions on JBoss EAP

For Use with Red Hat JBoss Enterprise Application Platform 7.2

## Legal Notice



## Abstract

This document provides information for administrators to troubleshoot transactions on JBoss EAP.

# Table of Contents

# CHAPTER 1. TRANSACTIONS IN JBOSS EAP

A transaction consists of two or more operations that must either all succeed or all fail. A successful outcome results in a commit, and a failed outcome results in a rollback. In a rollback, each member's state is reverted before the transaction attempts the commit.

## 1.1. TRANSACTION SUBSYSTEM

The **transactions** subsystem allows you to configure the Transaction Manager (TM) options, such as timeout values, transaction logging, statistics collection, and whether to use Java Transaction Service (JTS). The **transactions** subsystem consists of four main elements:

**Core environment**

The core environment includes the TM interface that allows the JBoss EAP server to control transaction boundaries on behalf of the resource being managed. A transaction coordinator manages communication with the transactional objects and resources that participate in transactions.

**Recovery environment**

The recovery environment of the JBoss EAP transaction service ensures that the system applies the results of a transaction consistently to all the resources affected by the transaction. This operation continues even if any application process or the machine hosting them crashes or loses network connectivity.

**Coordinator environment**

The coordinator environment defines custom properties for the transaction, such as default timeout and logging statistics.

**Object store**

JBoss EAP transaction service uses an object store to record the outcomes of transactions in a persistent manner for failure recovery. The Recovery Manager scans the object store and other locations of information, for transactions and resources that might need recovery.

## 1.2. PROPERTIES OF THE TRANSACTION

The typical standard for a well-designed transaction is that it is atomic, consistent, isolated, and durable (ACID):

**Atomic**

All members of the transaction must make the same decision regarding committing or rolling back the transaction.

**Consistent**

Transactions produce consistent results and preserve application specific invariants.

**Isolation**

The data being operated on must be locked before modification to prevent processes outside the scope of the transaction from modifying the data.

**Durable**

The effects of a committed transaction are not lost, except in the event of a catastrophic failure.

## 1.3. COMPONENTS OF A TRANSACTION

**Transaction Coordinator**

The coordinator governs the outcome of a transaction. It is responsible for ensuring that the web services invoked by the client arrive at a consistent outcome.

**Transaction Context**

Transaction context is the information about a transaction that is propagated, which allows the transaction to span multiple services.

**Transaction Participant**

Participants are the services enrolled in a transaction using a participant model.

**Transaction Service**

Transaction service captures the model of the underlying transaction protocol and coordinates with the participants affiliated with a transaction according to that model.

**Transaction API**

Transaction API provides an interface for transaction demarcation and the registration of participants.

## 1.4. PRINCIPLES OF TRANSACTION MANAGEMENT

### 1.4.1. XA Versus Non-XA Transactions

Non-XA transactions involve only one resource. They do not have a transaction coordinator and a single resource does all the transaction work. They are sometimes called local transactions.

XA transactions involve multiple resources. They also have a coordinating transaction manager with one or more databases, or other resources like JMS, all participating in a single transaction. They are referred to as global transactions.

# CHAPTER 2. CONFIGURING TRANSACTIONS

## 2.1. UNIQUE NODE IDENTIFIER

Unique node identifier allows JBoss EAP to recover transactions and transaction states that match only the specified node identifier. You can set the node identifier using the **com.arjuna.ats.arjuna.nodeIdentifier** property.

### 2.1.1. Importance of the Unique Node Identifier

When running XA recovery, you must configure the **Xid** types that JBoss EAP transactions can recover. Each **Xid** has the unique node identifier encoded in it and JBoss EAP only recovers the transactions and transaction states that match the specified node identifier.

You can configure the node identifier using the **JTAEnvironmentBean.xaRecoveryNodes** property, which can include multiple values in a list.

**WARNING**

A value of asterisk **"*"** forces JBoss EAP to recover, and possibly rollback, all the transactions irrespective of their node identifiers. It must be used with caution.

The value of **com.arjuna.ats.jta.xaRecoveryNode** property must be alphanumeric and must match the value of the **com.arjuna.ats.arjuna.nodeIdentifier** property.

## 2.2. CONFIGURING THE TRANSACTION MANAGER

You can configure the transaction manager using the web-based management console or the command line management CLI.

**Configuring the Transaction Manager Using the Management Console**
The following steps explain how to configure the transaction manager using the web-based management console:

1. Select the **Configuration** tab from the top of the screen.
2. If you are running JBoss EAP as a managed domain, choose the desired profile to modify.
3. From the **Subsystem** list, select **Transaction** and click **View**.
4. Select the appropriate tab for the settings that you want to configure, such as **Recovery** for recovery options.
5. Click **Edit**, make the necessary changes, and click **Save** to save the changes.

**Configuring the Transaction Manager Using the Management CLI**
Using the management CLI, you can configure the transaction manager using a series of commands. The commands all begin with /**subsystem=transactions** for a standalone server or /**profile=default**/**subsystem=transactions**/ for the **default** profile in a managed domain.

For a detailed listing of all the transaction manager configuration options, see the Transaction Manager Configuration Options for JBoss EAP.

## 2.3. CONFIGURING THE TRANSACTION MANAGER USING SYSTEM PROPERTIES

You can use either the management console, management CLI, or the system properties to configure many of the Transaction Manager options. However, the following options are configurable only using system properties. They are not configurable using the management CLI or management console.

| Property Name | Description |
| --- | --- |
| RecoveryEnvironmentBean.periodicRecoveryPeriod | Interval between recovery attempts, in seconds.<br>• Must be a positive Integer.<br>• Default is 120 seconds (2 minutes). |
| RecoveryEnvironmentBean.recoveryBackoffPeriod | Interval between the first and second recovery passes, in seconds.<br>• Must be a positive Integer.<br>• Default is 10 seconds. |
| RecoveryEnvironmentBean.periodicRecoveryInitializationOffset | Interval before first recovery pass, in seconds.<br>• Must be 0 or a positive Integer.<br>• Default is 0 seconds. |
| RecoveryEnvironmentBean.expiryScanInterval | Interval between expiry scans, in hours.<br>• Can be any Integer.<br>• 0 disables scanning.<br>• Negative values postpone the first run.<br>• Default is 12 hours. |

This example shows how to configure these system properties in the **standalone.xml** server configuration file.

```
<system-properties>
    <property name="RecoveryEnvironmentBean.periodicRecoveryPeriod" value="180"/>
    <property name="RecoveryEnvironmentBean.recoveryBackoffPeriod" value="20"/>
    <property name="RecoveryEnvironmentBean.periodicRecoveryInitializationOffset" value="5"/>
    <property name="RecoveryEnvironmentBean.expiryScanInterval" value="24"/>
</system-properties>
```

For more information on how to configure system properties, see System Properties in the *Configuration Guide*.

## 2.4. CONFIGURING YOUR DATASOURCE TO USE JTA

This task shows you how to enable Java Transaction API (JTA) on your datasource.

### Prerequisites

- Your database must support JTA. For more information, see the documentation from your database vendor.
- Create a non-XA datasource. For instructions, see the *Configuration Guide*.

**NOTE**

XA datasources, described in the *Configuration Guide*, are JTA capable by default.

### Configuring the Datasource to use JTA

1. Use the following management CLI command to set the **jta** attribute to **true**.

```
/subsystem=datasources/data-source=DATASOURCE_NAME:write-
attribute(name=jta,value=true)
```

**NOTE**

In a managed domain, precede this command with **/profile=*PROFILE_NAME***.

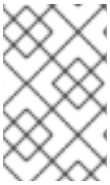
2. Reload the server for the changes to take effect.

```
reload
```

Your datasource is now configured to use JTA.

## 2.5. CONFIGURING THE ORB FOR JTS TRANSACTIONS

In a default installation of JBoss EAP, the Object Request Broker (ORB) support for transactions is disabled. You can configure ORB settings in the **iiop-openjdk** subsystem using the management CLI or the management console.



**NOTE**

The **iiop-openjdk** subsystem is available when using the *full* or *full-ha* profile in a managed domain, or the **standalone-full.xml** or **standalone-full-ha.xml** configuration file for a standalone server.

For a listing of the available configuration options for the **iiop-openjdk** subsystem, see IIOP Subsystem Attributes in the *Configuration Guide*.

### Configure the ORB Using the Management CLI

You can configure each aspect of the ORB using the management CLI. This is the minimum configuration for the ORB to be used with JTS.

You can configure the following management CLI commands for a managed domain using the **full** profile. If necessary, change the profile to suit the one you need to configure. If you are using a standalone server, omit the /**profile=full** portion of the commands.

**Enable the Security Interceptors**
Enable the **security** attribute by setting the value to **identity**.

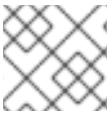
```
/profile=full/subsystem=iiop-openjdk:write-attribute(name=security,value=identity)
```

**Enable Transactions in the IIOP Subsystem**
To enable the ORB for JTS, set the value of **transactions** attribute to **full**, rather than the default **spec**.

```
/profile=full/subsystem=iiop-openjdk:write-attribute(name=transactions, value=full)
```

**Enable JTS in the Transactions Subsystem**

```
/profile=full/subsystem=transactions:write-attribute(name=jts,value=true)
```



NOTE

For JTS activation, the server must be restarted as reload is not enough.

### Configure the ORB Using the Management Console

1. Select the **Configuration** tab from the top of the management console. In a managed domain, you must select the appropriate profile to modify.
2. Select **Subsystems → IIOP (OpenJDK)** and click **View**.
3. Click **Edit** and modify the attributes as needed.
4. Click **Save** to save the changes.

# CHAPTER 3. MANAGING TRANSACTIONS

## 3.1. BROWSING TRANSACTIONS

The management CLI supports the ability to browse and manipulate transaction records. This functionality is provided by the interaction between the TM and the management API of JBoss EAP.

The Transaction Manager stores information about each pending transaction and the participants involved the transaction, in a persistent storage called the object store. The management API exposes the object store as a resource called the **log-store**. The **probe** operation reads the transaction logs and creates a node path for each record. You can call the **probe** operation manually, whenever you need to refresh the **log-store**. It is normal for transaction logs to appear and disappear quickly.

### Refreshing the Log Store

The following command refreshes the log store for server groups which use the profile **default** in a managed domain. For a standalone server, remove the **profile=default** from the command.

```
/profile=default/subsystem=transactions/log-store=log-store:probe
```

### Viewing All Prepared Transactions

To view all prepared transactions, first refresh the log store, then run the following command, which functions similarly to a file system **ls** command.

```
ls /profile=default/subsystem=transactions/log-store=log-store/transactions
```

Or

```
/host=master/server=server-one/subsystem=transactions/log-store=log-store:read-children-names(child-type=transactions)
```

Each transaction is shown, along with its unique identifier. Individual operations can be run against an individual transaction. For more information, see Administering a Transaction.

## 3.2. ADMINISTERING A TRANSACTION

### Viewing the Attributes of a Transaction

To view information about a transaction, such as its JNDI name, EIS product name and version, or its status, use the **read-resource** operation.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:read-resource
```

### Viewing the Details of a Transaction Participant

Each transaction log contains a child element called **participants**. Use the **read-resource** operation on this element to see the details of a participant of the transaction. Participants are identified by their JNDI names.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9/participants=java\:\/JmsXA:read-resource
```

The result may look similar to this:

```
{
    "outcome" => "success",
    "result" => {
        "eis-product-name" => "ActiveMQ",
        "eis-product-version" => "2.0",
        "jndi-name" => "java:/JmsXA",
        "status" => "HEURISTIC",
        "type" => "/StateManager/AbstractRecord/XAResourceRecord"
    }
}
```

The outcome status shown here is in a **HEURISTIC** state and is eligible for recovery. See Recover a Transaction Participant for more details.

In special cases it is possible to create orphan records in the object store, that is XAResourceRecords, which do not have any corresponding transaction record in the log. For example, XA resource prepared but crashed before the TM recorded and is inaccessible for the domain management API. To access such records you need to set management option **expose-all-logs** to **true**. This option is not saved in management model and is restored to **false** when the server is restarted.

```
/profile=default/subsystem=transactions/log-store=log-store:write-attribute(name=expose-all-logs,
value=true)
```

You can use this alternate command, which shows participant IDs of transaction in an aggregated form.

```
/host=master/server=server-one/subsystem=transactions/log-store=log-
store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9:read-children-names(child-type=participants)
```

### Deleting a Transaction Participant

Each transaction log supports a **delete** operation, to delete the transaction log representing the transaction.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9:delete
```

This deletes all participants in the transaction as well.

**WARNING**

Typically, you would leave participant log management to the recovery system or to the transaction log that owns it, but the **delete** operation is available for cases when you know it is safe to do so. In the case of heuristically completed XA resources, a **forget** call is triggered so that XA resource vendor logs are cleaned correctly. If this **forget** call fails, by default the **delete** operation will still succeed. You can override this behavior by setting the **ObjectStoreEnvironmentBean.ignoreMBeanHeuristics** system property to **false**.

### Recovering a Transaction Participant

Each transaction participant supports recovery by using the **recover** operation.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9/participants=2:recover
```

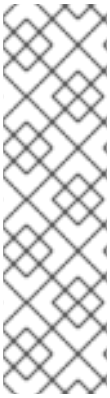
If the transaction participant's status is **HEURISTIC**, the **recover** operation switches the status to **PREPARE** and asks the periodic recovery process to replay the commit.

If the commit is successful, the participant is removed from the transaction log. You can verify this by running the **probe** operation on the **log-store** and checking that the participant is no longer listed. If this is the last participant, the transaction is also deleted.

### Refreshing the Status of a Transaction Participant

If a transaction needs recovery, you can use the **refresh** operation to be sure it still requires recovery, before attempting the recovery.

```
/profile=default/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-b66efc2\:4f9e6f8f\:9/participants=2:refresh
```



NOTE

In JBoss EAP 7.0, transaction failure exceptions are simply serialized and passed over the wire to the client. The client gets a **ClassNotFoundException** exception if they do not have the exception class on their class path.

JBoss EAP 7.1 introduced the **org.wildfly.common.rpc.RemoteExceptionCause** exception, which is known to the client as it is from the **wildfly** library. The server clones the original exception to this new one, puts all field of the original exception to a string form and adds them to the exception's message. The server then passes only exceptions of type **RemoteExceptionCause** to the client.

## 3.3. VIEWING TRANSACTION STATISTICS

If transaction manager statistics are enabled, you can view statistics on processed transactions by the transaction manager. See the Configuring the Transaction Manager section of the JBoss EAP *Configuration Guide* for information about how to enable transaction manager statistics.

You can view statistics using either the management console or the management CLI. In the management console, transaction statistics are available by navigating to the **Transaction** subsystem from the **Runtime** tab. From the management CLI, you can view statistics by using **include-runtime=true** to the **read-resource** operation. For example:

```
/subsystem=transactions:read-resource(include-runtime=true)
```

The following table shows the management console display name, management CLI attribute, and description for each transaction statistic.

**Table 3.1. Transactions Subsystem Statistics**

| Display Name | Attribute | Description |
|---|---|---|
| Aborted | number-of-aborted-transactions | The number of aborted transactions. |

| Display Name | Attribute | Description |
|---|---|---|
| Application Failures | number-of-application-rollbacks | The number of failed transactions, including timeouts, whose failure origin was an application. |
| Average Commit Time | average-commit-time | The average time of transaction commit, in nanoseconds, measured from when the client calls commit until the transaction manager determines that it was successful. |
| Committed | number-of-committed-transactions | The number of committed transactions. |
| Heuristics | number-of-heuristics | The number of transactions in a heuristic state. |
| Inflight Transactions | number-of-inflight-transactions | The number of transactions which have begun but not yet terminated. |
| Nested Transactions | number-of-nested-transactions | The total number of nested transactions created. |
| Number of Transactions | number-of-transactions | The total number of transactions created, including nested. |
| Resource Failures | number-of-resource-rollbacks | The number of failed transactions whose failure origin was a resource. |
| System Failures | number-of-system-rollbacks | The number of transactions that have been rolled back due to internal system errors. |
| Timed Out | number-of-timed-out-transactions | The number of transactions that have rolled back due to timeout. |

## 3.4. CONFIGURING THE TRANSACTIONS OBJECT STORE

Transactions need a place to store objects. One of the options for object storage is a JDBC datasource. If performance is a particular concern, the JDBC object store can be slower than a file system or ActiveMQ journal object store.

IMPORTANT

If the **transactions** subsystem is configured to use Apache ActiveMQ Artemis journal as storage type for transaction logs, then two instances of JBoss EAP are not permitted to use the same directory for storing the journal. Application server instances can not share the same location and each has to configure a unique location for it.

NOTE

Losing a transaction object store can lead to losing data consistency. Thus, the object store needs to be placed on a *safe* drive.

### Using a JDBC Datasource as a Transactions Object Store

Follow the below steps to use a JDBC datasource as a transactions object store.

1. Create a datasource, for example, **TransDS**. For instructions on a non-XA datasource, see the Create a Non-XA datasource section of the JBoss EAP *Configuration Guide*. Note that the datasource's JDBC driver must be installed as a core module , as described in the JBoss EAP *Configuration Guide*, not as a JAR deployment, for the object store to work properly.

2. Set the datasource's **jta** attribute to **false**.

```
/subsystem=datasources/data-source=TransDS:write-attribute(name=jta, value=false)
```

3. Set the **jdbc-store-datasource** attribute to the JNDI name for the datasource to use, for example, **java:jboss/datasources/TransDS**.

```
/subsystem=transactions:write-attribute(name=jdbc-store-datasource,
value=java:jboss/datasources/TransDS)
```

4. Set the **use-jdbc-store** attribute to **true**.

```
/subsystem=transactions:write-attribute(name=use-jdbc-store, value=true)
```

5. Restart the JBoss EAP server for the changes to take effect.

### Transactions JDBC Store Attributes

The following table identifies all of the available attributes related to JDBC object storage.

NOTE

Attribute names in this table are listed as they appear in the management model, for example, when using the management CLI. See the schema definition file located at ***EAP_HOME*/docs/schema/wildfly-txn_4_0.xsd** to view the elements as they appear in the XML, as there may be differences from the management model.

**Table 3.2. JDBC Store Attributes for Transactions**

| Property | Description |
|---|---|
| use-jdbc-store | Set this to **true** to enable the JDBC store for transactions. |
| jdbc-store-datasource | The JNDI name of the JDBC datasource used for storage. |
| jdbc-action-store-drop-table | Whether to drop and recreate the action store tables at launch. The default is **false**. |
| jdbc-action-store-table-prefix | The prefix for the action store table names. |

| Property | Description |
| --- | --- |
| jdbc-communication-store-drop-table | Whether to drop and recreate the communication store tables at launch. The default is **false**. |
| jdbc-communication-store-table-prefix | The prefix for the communication store table names. |
| jdbc-state-store-drop-table | Whether to drop and recreate the state store tables at launch. The default is **false**. |
| jdbc-state-store-table-prefix | The prefix for the state store table names. |

### Using the ActiveMQ Journal Object Store

Follow the below steps to use an ActiveMQ journal object store.

1. Set the **use-journal-store** attribute to **true**.

```
/subsystem=transactions:write-attribute(name=use-journal-store,value=true)
```

2. Restart the JBoss EAP server for the changes to take effect.

# CHAPTER 4. MONITORING TRANSACTIONS

## 4.1. CONFIGURING LOGGING FOR THE TRANSACTIONS SUBSYSTEM

You can control the amount of information logged about transactions, independent of other logging settings in JBoss EAP. You can configure the logging settings using the management console or the management CLI.

### Configuring the Transaction Logger Using the Management Console

1. Navigate to the **Logging** subsystem configuration.
   a. In the management console, click the **Configuration** tab. If you use a managed domain, you must choose the appropriate server profile.
   b. Select **Subsystems → Logging → Configuration** and click **View**.
2. Edit the **com.arjuna** attributes.
   Select the **Categories** tab. The **com.arjuna** entry is already present. Select **com.arjuna** and click **Edit**. You can change the log level and choose whether to use parent handlers or not.
   - Log Level:
     As transactions can produce a lot of logging output, the default logging level is set to **WARN** so that the server log is not overwhelmed by transaction output. If you need to check transaction processing details, use the **TRACE** log level so that transaction IDs are shown.
   - Use Parent Handlers:
     Parent handler indicates whether the logger should send its output to its parent logger. The default behavior is **true**.
3. Click **Save** to save the changes.

### Configuring the Transaction Logger Using the Management CLI

Use the following command to set the logging level from the management CLI. For a standalone server, remove the /**profile=default** from the command.

```
/profile=default/subsystem=logging/logger=com.arjuna:write-attribute(name=level,value=VALUE)
```

### 4.1.1. Enabling the TRACE Log Level

The **TRACE** level logging allows you diagnose JCA issues in JBoss EAP. You can execute the following command to enable the **TRACE** level logging for **com.arjuna** class:

```
/profile=default/subsystem=logging/logger=com.arjuna:write-attribute(name=level,value=TRACE)
```

For a standalone server, remove the /**profile=default** from the command.

### 4.1.2. Enabling the Transaction Bridge Logger

The transaction bridge is a layer on top of the XTS and JTA/JTS components of the Transaction Manager. It interacts with other parts of JBoss EAP server. You can enable verbose logging of these components that interact with the Transaction Manager for a detailed explanation of the system's operations.

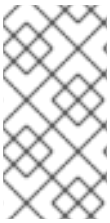
The transaction bridge uses the **logging** subsystem. When running the JBoss EAP server, logging is configured from the **logging** subsystem configuration in the **standalone-xts.xml** file. Logging for the transaction bridge is useful for debugging purposes.

You can use the following management CLI command to configure the **org.jboss.jbossts.txbridge** logger to enable the transaction bridge logging:

```
/subsystem=logging/logger=org.jboss.jbossts.txbridge:add(level=ALL)
```

This generates the following XML in the server configuration file:

```
<logger category="org.jboss.jbossts.txbridge">
  <level name="ALL" />
</logger>
```



NOTE

Deployment ordering issues may result in the Transaction Manager components becoming active before the **logging** subsystem is fully configured, including the transaction bridge. In such cases a default logging level gets applied during startup, thereby resulting in detailed debug messages being missed.

You can configure the **com.arjuna** logger to enable verbose logging using the following management CLI command:

```
/subsystem=logging/logger=com.arjuna:write-attribute(name=level,value=ALL)
```

This generates the following XML in the server configuration file:

```
<logger category="com.arjuna">
  <level name="ALL" />
</logger>
```

### 4.1.3. Transaction Log Messages

You can track the transaction status while keeping the log files readable by using the **DEBUG** log level for the transaction logger. For detailed debugging, use the **TRACE** log level. Refer to Configuring Logging for the Transactions Subsystem for information on configuring the transaction logger.

Transaction Manager (TM) can generate a lot of logging information when configured to log in the **TRACE** log level. Following are some of the most commonly-seen messages. This list is not comprehensive, so you may see messages other than these.

**Table 4.1. Transaction State Change**

| | |
|---|---|
| Transaction Begin | When a transaction begins, a method **Begin** of class **com.arjuna.ats.arjuna.coordinator.BasicAction** is executed and presented in the log with the message **BasicAction::Begin() for action-id <transaction uid>**. |

| | |
|---|---|
| Transaction Commit | When a transaction commits, a method **Commit** of class **com.arjuna.ats.arjuna.coordinator.BasicAction** is executed and presented in the log with the message **BasicAction::Commit() for action-id <transaction uid>**. |
| Transaction Rollback | When a transaction rolls back, a method **Rollback** of class **com.arjuna.ats.arjuna.coordinator.BasicAction** is executed and presented in the log with the message **BasicAction::Rollback() for action-id <transaction uid>**. |
| Transaction Timeout | When a transaction times out, a method **doCancellations** of **com.arjuna.ats.arjuna.coordinator.TransactionReaper** is executed and presented in log as **Reaper Worker <thread id> attempting to cancel <transaction uid>**. You will then see the same thread rolling back the transaction as shown above. |

### 4.1.4. Decoding Transaction Log Files

#### 4.1.4.1. Locating the XID/UID of a Transaction

The **javax.transaction.TransactionManager** interface provides two ways to locate the transaction identifier:

- You can call the **toString** method to print complete information about the transaction, including the identifier.
- Alternatively, you can cast the **javax.transaction.Transaction** instance to a **com.arjuna.ats.jta.transaction.Transaction** and then call either the **get_uid** method, which returns the ArjunaCore Uid representation, or call the **getTxId** method, which returns the Xid for the global identifier, that is not the branch qualifier.

```
com.arjuna.ats.jta.transaction.Transaction arjunaTM =
(com.arjuna.ats.jta.transaction.Transaction)tx.getTransaction();
System.out.println("Transaction UID" +arjunaTM.get_uid());
```

#### 4.1.4.2. Finding the Transaction Status and Resources

**TransactionStatusConnectionManager**
The **TransactionStatusConnectionManager** object is used by the recovery modules to retrieve the status of the transaction. It acts like a proxy for **TransactionStatusManager** objects by maintaining a table of **TransactionStatusConnector** objects, each of which connects to a **TransactionStatusManager** object in the application process.

You can retrieve the transaction status using the **getTransactionStatus** method that takes a transaction Uid and, if available, a transaction type as parameters.

1. The process Uid field in the transactions Uid parameter is used to lookup the target **TransactionStatusManagerItem** host-port pair in the transaction object store.
2. The host-port pair is used to make a TCP connection to the target **TransactionStatusManager** object by using a **TransactionStatusConnector** object.

3. The **TransactionStatusConnector** passes the transaction Uid and the transaction type to the **TransactionStatusManager** in order to retrieve the status of the transactions.

The following code example shows how to retrieve the **TransactionStatusConnectionManager** and check the transaction status:

```
// Transaction id
Uid tx = new Uid();
. . . .
TransactionStatusConnectionManager tscm = new TransactionStatusConnectionManager();

// Check if the transaction aborted
assertEquals(tscm.getTransactionStatus(tx), ActionStatus.ABORTED);
```

**TransactionStatusManager**
The **TransactionStatusManager** object acts as an interface for the Recovery Manager to obtain the status of transactions from the running application processes. One **TransactionStatusManager** per application process is created by the **com.arjuna.ats.arjuna.coordinator.TxControl** class. A TCP connection is used for communication between the Recovery Manager and **TransactionStatusManager**. Any free port is used by the **TransactionStatusManager** by default. However, the port used can be fixed using the following the property:

```
$ EAP_HOME/bin/standalone.sh -
DRecoveryEnvironmentBean.transactionStatusManagerPort=NETWORK_PORT_NUMBER
```

1. On creation, the **TransactionStatusManager** obtains a port that is stored with the host in the object store as a **TransactionStatusManagerItem**.

2. A **Listener** thread is started which waits for a connection request from the **TransactionStatusConnector**.

3. When the connection is established, a **Connection** thread is created that runs the **AtomicActionStatusService** service. This service accepts a transaction Uid and a transaction type, if available, from the **TransactionStatusConnector** object.

4. The transaction status is obtained from the local transaction table and returned back to the **TransactionStatusConnector** object.

#### 4.1.4.3. Viewing the Transaction History

By default, the transaction service does not maintain any history about the transactions. However, you can set the **CoordinatorEnvironmentBean.enableStatistics** property variable to **true** for the transaction service to maintain information about the number of transactions created and their respective outcomes.

You can use the following management CLI command to enable the statistics:

```
/subsystem=transactions:write-attribute(name=enable-statistics,value=true)
```

You can obtain more detailed transaction statistics programmatically by using the **com.arjuna.ats.arjuna.coordinator.TxStats** class.

**Example: TxStats Class**

```
public class TxStats
```

```
{
    /**
     * @return the number of transactions (top-level and nested) created so far.
     */

    public static int numberOfTransactions();

    /**
     * @return the number of nested (sub) transactions created so far.
     *

     public static int numberOfNestedTransactions();

     /**
     * @return the number of transactions which have terminated with heuristic
     *         outcomes.
     */

    public static int numberOfHeuristics();
    /**
     * @return the number of committed transactions.
     */

    public static int numberOfCommittedTransactions();

    /**
     * @return the total number of transactions which have rolled back.
     */

    public static int numberOfAbortedTransactions();

    /**
     * @return total number of inflight (active) transactions.
     */

    public static int numberOfInflightTransactions ();

    /**
     * @return total number of transactions rolled back due to timeout.
     */

    public static int numberOfTimedOutTransactions ();
    /**
     * @return the number of transactions rolled back by the application.
     */

    public static int numberOfApplicationRollbacks ();

    /**
     * @return number of transactions rolled back by participants.
     */

    public static int numberOfResourceRollbacks ();

    /**
     * Print the current information.
```

```
     */

    public static void printStatus(java.io.PrintWriter pw);
}
```

The **com.arjuna.ats.arjuna.coordinator.ActionManager** class provides further information about specific active transactions using the **getNumberOfInflightTransactions** method that returns the list of currently active transactions.

# CHAPTER 5. HANDLING TRANSACTION MANAGER EXCEPTIONS

## 5.1. DEBUGGING A TIMED-OUT TRANSACTION

There can be many reasons for a transaction timeout, such as:

- Slow server performance
- Thread is stuck waiting for something or hangs up
- Thread needs more than the configured transaction timeout time to complete the processing

You can look at the logs for following error message to identify a timed-out transaction:

```
WARN ARJUNA012117 "TransactionReaper::check timeout for TX {0} in state {1}"
```

where ***{0}*** is the Uid of the transaction and ***{1}*** is the transaction manager's view of the state {1} of the timed-out transaction.

Transaction Manager provides the following options to debug the transaction timeouts:

- You can configure timeout values for transactions to control the transaction lifetimes. The **transactions** subsystem rolls back the transaction if the timeout value elapses before a transaction terminates because of committing or rolling back.
- You can use the **setTransactionTimeout** method of the **XAResource** interface to propagate the current transaction to the resource manager. If supported, this operation overrides any default timeout associated with the resource manager. Overriding the timeout is useful in situations like the following:
  - when long-running transactions have lifetimes that exceed the default
  - when using the default timeout might cause the resource manager to roll back before the transaction terminates, causing the transaction to roll back as well.
- If you do not specify a timeout value or use a value of **0**, transaction manager uses an implementation-specific default value. In JBoss EAP transaction manager, the **CoordinatorEnvironmentBean.defaultTimeout** property represents this implementation-specific default value. The default value is **300** seconds. A value of **0** disables the default transaction timeouts.
  You can modify the default transaction timeout using the following management CLI command:

  ```
  /subsystem=transactions:write-attribute(name=default-timeout,value=VALUE)
  ```

  When running in a managed domain, you must specify which profile to update by preceding the command with /**profile=*PROFILE_NAME***
- JBoss EAP Transaction Manager supports an all-or-nothing approach to call the **setTransactionTimeout** method on the **XAResource** instances. You can set the **JTAEnvironmentBean.xaTransactionTimeoutEnabled** property to **true**, which is the default, to call the method on all the instances. Otherwise, you can use the

**setXATransactionTimeoutEnabled** method of the **com.arjuna.ats.jta.common.JTAEnvironmentBean** class to disable timeout and specify them on a per-transaction basis.

## 5.2. MIGRATING LOGS TO A NEW JBOSS EAP SERVER

### Prerequisites

Ensure that the **transactions** subsystem is configured identically between the old and the new JBoss EAP. This identical configuration includes the list of JTA datasources because recovering the logs needs to contact the datasources used by any of the recovered logs.

### 5.2.1. Migrating the File-based Log Storage

To migrate the transaction manager logs to a new JBoss EAP server, you can copy the logs to the new JBoss EAP server.

You can use the following commands to copy the file-based logs:

1. Browse to your ***EAP_HOME*** directory.
2. Create an archive of the logs using the following command:

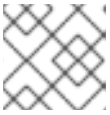
```
$ tar -cf logs.tar ./standalone/data/tx-object-store
```

3. Extract the archived logs to the new ***EAP_HOME*** directory using the following command:

```
$ tar -xf logs.tar -C NEW_EAP_HOME
```

### 5.2.2. Migrating the JDBC Store-based Log Storage

- You can configure the new JBoss EAP server to use the old database and tables as described in Using a JDBC Datasource as a Transactions Object Store .
- Alternatively, you can determine the database and the tables used for the transaction logs. Then, you can use an SQL tool to back up the tables and restore them to the new database.



**NOTE**

You can find an SQL query tool in the **h2** JAR file shipped with JBoss EAP.

## 5.3. ENABLING XTS ON JBOSS EAP

XML Transaction Service (XTS) component of the transaction manager supports the coordination of private and public web services in a business transaction. XTS provides WS-AT and WS-BA support for web services hosted on the JBoss EAP server. It is an optional subsystem, which can be enabled using the **standalone-xts.xml** configuration.

#### Starting JBoss EAP Server with XTS Enabled

1. Change to the JBoss EAP server directory:

```
cd $EAP_HOME
```

2. Copy the example XTS configuration file into the /**configuration** directory:

```
cp docs/examples/configs/standalone-xts.xml standalone/configuration
```

3. Start the JBoss EAP server, specifying the **xts** configuration:
   Linux:

```
bin/standalone.sh --server-config=standalone-xts.xml
```

   Windows:

```
bin\standalone.bat --server-config=standalone-xts.xml
```

## 5.4. CLEARING UP EXPIRED TRANSACTIONS

The following properties allow you to clear up the expired transactions:

**ExpiryEntryMonitor**

When the Recovery Manager initializes an expiry scanner thread, the **ExpiryEntryMonitor** object is created, which is used to remove dead items from the object store. A number of scanner modules are loaded dynamically, which removes the dead items for a particular type.
You can configure the scanner modules in the properties file using the **RecoveryEnvironmentBean.expiryScanners** system property. The scanner modules are loaded at the time of initialization.

```
$ EAP_HOME/bin/standalone.sh -
DRecoveryEnvironmentBean.expiryScanners=CLASSNAME1,CLASSNAME2
```

**expiryScanInterval**

All the scanner modules are called periodically to scan for dead items by the **ExpiryEntryMonitor** thread. You can configure this period, in hours, using the **expiryScanInterval** system property, as shown in the example below:

```
$ EAP_HOME/bin/standalone.sh -
DRecoveryEnvironmentBean.expiryScanInterval=EXPIRY_SCAN_INTERVAL
```

All scanner modules inherit the same behavior from the **ExpiryScanner** interface. This interface provides a scan method that is implemented by all the scanner modules, including the following. The scanner thread calls this scan method.

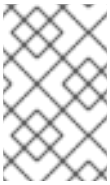
**ExpiredTransactionStatusManagerScanner**

The **ExpiredTransactionStatusManagerScanner** removes the dead **TransactionStatusManagerItems** from the object store. These items remain in the object store for a certain period before they are deleted, which is 12 hours by default. You can configure this time period, in hours, using the **transactionStatusManagerExpiryTime** system property as shown in the example below:

```
$ EAP_HOME/bin/standalone.sh -
DRecoveryEnvironmentBean.transactionStatusManagerExpiryTime=TRANSACTION_STATUS_MANAGER_EXPIRY_TIME
```

**AtomicActionExpiryScanner**

The **AtomicActionExpiryScanner** moves transaction logs for **AtomicActions** that are assumed to have completed. For example, if a failure occurs after a participant has been told to commit but before the **transactions** subsystem can update the logs, then upon recovery the JBoss EAP transaction manager attempts to replay the commit request. This replay will obviously fail, thus preventing the log from being removed. The **AtomicActionExpiryScanner** is also used when logs cannot be recovered automatically for reasons such as being corrupt or zero length. All logs are moved to a specific location based on the old location appended with /**Expired**.



NOTE

**AtomicActionExpiryScanner** is disabled by default. You can enable it by adding it to the transaction manager properties file. You need not enable it to cope with corrupt logs.

## 5.5. RECOVERING HEURISTIC OUTCOMES

A heuristic completion occurs when a transaction resource makes a one-sided decision, during the completion stage of a distributed transaction, to commit or rollback the transaction updates. This can leave distributed data in an indeterminate state. Network failures or resource timeouts are possible causes for heuristic completion. Heuristic completion throws one of the following heuristic outcome exceptions:

**HEURISTIC_COMMIT**

This exception is thrown when the transaction manager decides to rollback, but somehow all the resources had already committed on their own. In this case, you need not do anything because a consistent termination was reached.

**HEURISTIC_ROLLBACK**

This exception implies that the resources have all done a rollback because the commit decision from the transaction manager was delayed. Similar to **HEURISTIC_COMMIT**, in this case also you need not do anything because a consistent termination was reached.

**HEURISTIC_HAZARD**

This exception occurs when the disposition of some of the updates is unknown. For those that are known, they have either all been committed or all rolled back.

**HEURISTIC_MIXED**

This exception occurs when some parts of the transaction were rolled back while others were committed.

This procedure shows how to handle a heuristic outcome of a transaction using the Java Transaction API (JTA).

1. The cause of a heuristic outcome in a transaction is that a resource manager promised it could commit or rollback, and then failed to fulfill the promise. This could be due to a problem with a third-party component, the integration layer between the third-party component and JBoss EAP, or JBoss EAP itself.
   By far, the most common two causes of heuristic errors are transient failures in the environment and coding errors dealing with resource managers.

2. Usually, if there is a transient failure in your environment, you will know about it before you find out about the heuristic error. This could be due to a network outage, hardware failure, database failure, power outage, or a host of other things.

If you come across a heuristic outcome in a test environment during stress testing, it implies weaknesses in your test environment.

**WARNING**

JBoss EAP automatically recovers transactions that were in a non-heuristic state at the time of failure, but it does not attempt to recover the heuristic transactions.

3. If you have no obvious failure in your environment, or if the heuristic outcome is easily reproducible, it is probably due to a coding error. You must contact the third-party vendors to find out if a solution is available.
   If you suspect the problem is in the transaction manager of JBoss EAP itself, you must raise a support ticket.

4. You can attempt to recover the transaction manually using the management CLI. For instructions on manually recovering a transaction, see the Recovering a Transaction Participant section.

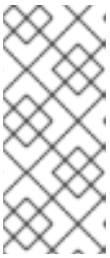
5. The process of resolving the transaction outcome manually is dependent on the exact circumstance of the failure. Perform the following steps, as applicable to your environment:

   a. Identify which resource managers were involved.

   b. Examine the state of the transaction manager and the resource managers.

   c. Manually force log cleanup and data reconciliation in one or more of the involved components.

6. In a test environment, or if you do not care about the integrity of the data, deleting the transaction logs and restarting JBoss EAP gets rid of the heuristic outcome. By default, the transaction logs are located in the ***EAP_HOME*/standalone/data/tx-object-store/** directory for a standalone server, or the ***EAP_HOME*/domain/servers/*SERVER_NAME*/data/tx-object-store/** directory in a managed domain. In the case of a managed domain, *SERVER_NAME* refers to the name of the individual server participating in a server group.



**NOTE**

The location of the transaction log also depends on the object store in use and the values set for the **object-store-relative-to** and **object-store-path** parameters. For file system logs, such as a standard shadow and Apache ActiveMQ Artemis logs, the default directory location is used, but when using a JDBC object store, the transaction logs are stored in a database.

### 5.5.1. Guidelines on Making Decisions for Heuristic Outcomes

**Problem Detection**
A heuristic decision is one of the most critical errors that can happen in a transaction system. It can lead to parts of the transaction being committed, while other parts are rolled back. Thus, it can violate the atomicity property of the transaction and can possibly lead to corruption of the data integrity.

A recoverable resource maintains all the information about the heuristic decision in stable storage until it is required by the transaction manager. The actual data saved in stable storage depends on the type of recoverable resource and is not standardized. You can parse through the data and possibly edit the resource to correct any data integrity problems.

Heuristic outcomes are stored in the server log and can be identified using the resource manager and transaction manager.

**Manually Committing or Rolling Back a Transaction**
Generally, you cannot manually commit or rollback a transaction. From the JBoss EAP transaction management perspective, you can move a transaction back to the pending list for automated recovery to try again or delete the record. For example:

You can use the **read-resource** operation to check the status of the participants in the transaction:

```
/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=2:read-resource
```

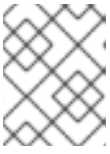
The result will look similar to this:

```
{
    "outcome" => "success",
    "result" => {
        "eis-product-name" => "ArtemisMQ",
        "eis-product-version" => "2.0",
        "jndi-name" => "java:/JmsXA",
        "status" => "HEURISTIC_HAZARD",
        "type" => "/StateManager/AbstractRecord/XAResourceRecord"
    }
}
```

The outcome status shown here is a **HEURISTIC_HAZARD** state and is eligible for recovery.

**Recovering the HEURISTIC_HAZARD Exception**
The following steps show an example of how to recover a **hazard** type heuristic outcome.

1. To begin the recovery, you must consult each resource manager and establish the outcomes of the various branches that are identifiable from the transaction manager tooling. However, you should not need to force a resource manager to commit or rollback. You must rather inspect the resource manager to know the state of the heuristic exception.
   The following are reference links for listing and resolving heuristic outcomes for various resource managers:



**NOTE**

These links are for reference purpose only and are subject to change. Please consult the vendor documentation for details.

- Manual resolution of in-doubt transactions in Oracle
- Manual resolution of in-doubt transactions in DB2
- View prepared transactions for two-phase commit in PostGreSQL and commit or rollback
- XA transaction syntax in MySQL

- XA transaction implementation in MariaDB

2. You must execute the recover operation, as shown in the following example:

```
/subsystem=transactions/log-store=log-store/transactions=0\:ffff7f000001\:-
b66efc2\:4f9e6f8f\:9/participants=2:recover
```

Running the **recover** operation changes the state of the transaction to **PREPARE** and triggers a recovery attempt by replaying the **commit** operation. If the recovery attempt is successful, the participant is removed from the transaction log.

You can verify this by running the **probe** operation on the **log-store** element again. The participant should no longer be listed. If this is the last participant, the transaction is also deleted.

**Recovering the HEURISTIC_ROLLBACK and HEURISTIC_COMMIT Exceptions**
If the heuristic outcome is a **rollback** type, then:

- The resource should not be able to commit the transaction, provided the resource manager is well implemented.
- You must decide whether you should delete the branch from the resource manager, using a forget call, so that the rest of the transaction can commit normally and be cleaned from the transaction store.
- If you do not delete the branch from the resource manager, then the transaction will remain in the transaction store forever.

On the other hand, if the heuristic outcome was a **commit** type, then you must use the business semantics to deal with the inconsistent outcome.

**Further Actions When Manual Reconciliation Fails**
You can check the database transaction table, which is the **DBA_2PC_PENDING** table for Oracle. However, these will depend upon the specific resource managers. Transaction Manager can provide you with the branches to inspect in each resource manager.

You should consult the vendor's documentation on this resource manager for details. If you suspect that the problem is caused by the third party resource manager, you must consider raising a support ticket with your supplier.

*Revised on 2019-09-26 10:42:42 UTC*