33
Application Continuity for Java

The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with a logon storm, which is a sudden increase in the number of client connection requests.

To overcome such problems, Oracle Database 12*c* Release 1 (12.1) introduced the Application Continuity feature that masks database outages to the application and end users are not exposed to such outages.

> **Note:**
>
> - You must use Transaction Guard for using Application Continuity.
> - Explicit switch of container and service in applications through the `ALTER SESSION SET CONTAINER` statement is not supported with Application Continuity.

Application Continuity provides a general purpose, application-independent solution that enables recovery of work from an application perspective, after the occurrence of a planned or unplanned outage. The outage can be related to system, communication, or hardware following a repair, a configuration change, or a patch application.

This chapter discusses the JDBC aspect of Application Continuity in the following sections:

- About Configuring Oracle JDBC for Application Continuity for Java
- About Configuring Oracle Database for Application Continuity for Java
- Application Continuity with DRCP
- Application Continuity Support for XA Data Source
- About Identifying Request Boundaries in Application Continuity for Java
- Support for Transparent Application Continuity
- Establishing the Initial State Before Application Continuity Replays
- About Delaying the Reconnection in Application Continuity for Java
- About Retaining Mutable Values in Application Continuity for Java
- Application Continuity Statistics
- About Disabling Replay in Application Continuity for Java

Related Topics

- Transaction Guard for Java

33.1 About Configuring Oracle JDBC for Application Continuity for Java

For configuring Oracle JDBC on the client-side, you must specify an Oracle JDBC driver data source, and set the necessary properties on this data source. The data source depends on the Oracle JDBC driver version that you use.

- For Oracle JDBC driver 23ai, Application Continuity is auto-enabled for all the driver data sources. To support Application Continuity, all you need is to connect to a database service with AC/TAC enabled.

  To disable Application Continuity on any driver data source, you can perform any of the following steps:

  - Specify `oracle.jdbc.enableACSupport=false` as a Java system property
  - Specify `oracle.jdbc.enableACSupport=false` as a driver connection property
  - Set `?oracle.jdbc.enableACSupport=false` in the URL
- For using Oracle JDBC driver 19c or older, you must use `oracle.jdbc.replay.OracleDataSourceImpl` for Application Continuity.

> **Note:**
>
> Regardless of the Oracle JDBC driver version, the configuration of any driver data source or associated properties (including connection properties) is usually specific to the application server, container, or framework that you are using, for example, WebLogic, WebSphere, JBoss, Spring, Tomcat, and so on. If you use a connection pool, then this data source is usually specified as a connection factory class for the connection pool.

> **See Also:**
>
> Application Continuity Support for XA Data Source, if you use XA transactions, or XA data source for local transactions.

The following code snippet illustrates the usage of `oracle.jdbc.datasource.impl.OracleDataSource` in a standalone JDBC application:

```
import java.sql.Connection;
import oracle.jdbc.datasource.impl.OracleDataSource;
......

{
  ......
  OracleDataSource ods = new OracleDataSource();
  ods.setUser(user);
  ods.setPassword(passwd);
  ods.setURL(url);
  ......  // Other data source configuration like callback, timeouts, etc.
```

```
  Connection conn = ods.getConnection();
  conn.beginRequest();  // Explicit request begin
  ......  // JDBC calls protected by Application Continuity
  conn.endRequest();  // Explicit request end
  conn.close();
}
```

You must remember the following points while using the connection URL:

- Always use the thin driver in the connection URL.
- Always connect to a service. Never use `instance_name` or SID because these do not direct to known good instances and SID is deprecated.
- If the addresses in the `ADDRESS_LIST` at the client does not match the `REMOTE_LISTENER` setting for the database, then it does not connect showing `services cannot be found`. So, the addresses in the `ADDRESS_LIST` at the client *must* match the `REMOTE_LISTENER` setting for the database:
  - If `REMOTE_LISTENER` is set to the `SCAN_VIP`, then the `ADDRESS_LIST` uses `SCAN_VIP`
  - If `REMOTE_LISTENER` is set to the host VIPs, then the `ADDRESS_LIST` uses the same host VIPs
  - If `REMOTE_LISTENER` is set to both `SCAN_VIP` and host VIPs, then the `ADDRESS_LIST` uses `SCAN_VIP` and the same host VIPs

  > **Note:**
  >
  > For Oracle clients prior to release 11.2, the `ADDRESS_LIST` must be upgraded to use SCAN, which means expanding the `ADDRESS_LIST` to three `ADDRESS` entries corresponding to the three SCAN IP addresses.
  >
  > If such clients connect to a database that is upgraded from an earlier release through Database AutoUpgrade, then you must retain the `ADDRESS_LIST` of these clients set to the HOST VIPs. However, if `REMOTE_LISTENER` is changed to `ONLY SCAN`, or the clients are moved to a newly installed Oracle Database 12*c* Release 1, where `REMOTE_LISTENER` is `ONLY SCAN`, then they do not get a complete service map, and may not always be able to connect.

- Set `RETRY_COUNT`, `RETRY_DELAY`, `CONNECT_TIMEOUT`, and `TRANSPORT_CONNECT_TIMEOUT` parameters in the connection string. This is a general recommendation for configuring the JDBC thin driver connections, starting from Oracle Database Release 12.1.0.2. These settings improve acquiring new connections at run time, at replay, and during work drains for planned outages.

  The `CONNECT_TIMEOUT` parameter is equivalent to the `SQLNET.OUTBOUND_CONNECT_TIMEOUT` parameter in the `sqlnet.ora` file and applies to the full connection. The `TRANSPORT_CONNECT_TIMEOUT` parameter applies as per the `ADDRESS` parameter. If the service is not registered for a failover or restart, then retrying is important when you use SCAN. For example, for using remote listeners pointing to SCAN addresses, you should use the following settings:

> **Note:**
>
> In the following code snippets, the `TRANSPORT_CONNECT_TIMEOUT` parameter is set to 3. However, if you are using 12.2.0.1 JDBC driver in your application, without the required patch that fixes bug 25977056, then you must set the `TRANSPORT_CONNECT_TIMEOUT` parameter to 3000.

```
jdbc:oracle:thin:@(DESCRIPTION =
 (TRANSPORT_CONNECT_TIMEOUT=3)
 (RETRY_COUNT=20)(RETRY_DELAY=3s)(FAILOVER=ON)
 (ADDRESS_LIST =(ADDRESS=(PROTOCOL=tcp)
  (HOST=CLOUD-SCANVIP.example.com)(PORT=5221))
 (CONNECT_DATA=(SERVICE_NAME=orcl)))

REMOTE_LISTENERS=CLOUD-SCANVIP.example.com:5221
```

Similarly, for using remote listeners pointing to VIPs at the database, you should use the following settings:

```
jdbc:oracle:thin:@(DESCRIPTION =
(TRANSPORT_CONNECT_TIMEOUT=3)
(CONNECT_TIMEOUT=60)(RETRY_COUNT=20)(RETRY_DELAY=3s)(FAILOVER=ON)
 (ADDRESS_LIST=
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP1.example.com)(PORT=5221) )
(ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP2.example.com)(PORT=5221) )
 (ADDRESS=(PROTOCOL=tcp)(HOST=CLOUD-VIP3.example.com)(PORT=5221) ))
(CONNECT_DATA=(SERVICE_NAME=orcl)))

REMOTE_LISTENERS=CLOUD-VIP1.example.com:5221
```

> **See Also:**
>
> - *Oracle Database Net Services Reference* for more information about local naming parameters
> - *Oracle Real Application Clusters Administration and Deployment Guide*

Related Topics

- Data Sources and URLs

33.1.1 Support for Concrete Classes with Application Continuity

Starting from Oracle Database Release 18c, the JDBC driver supports concrete classes with Application Continuity.

The classes are the following:

- `oracle.sql.CLOB`
- `oracle.sql.NCLOB`
- `oracle.sql.BLOB`
- `oracle.sql.BFILE`
- `oracle.sql.STRUCT`

- `oracle.sql.REF`
- `oracle.sql.ARRAY`

The only concrete classes in the `oracle.sql` package, which are not supported with Application Continuity, are:

- `oracle.sql OPAQUE`
- `oracle.sql.ANYDATA`
- `oracle.sql.JAVA_STRUCT`

Although the Oracle JDBC driver supports the concrete classes present in the `oracle.sql` package with Application Continuity, Oracle strongly recommends that you update your applications to use the Java interfaces instead. These interfaces can be the standard JDBC `java.sql` interfaces, or Oracle JDBC extension interfaces in the `oracle.jdbc` package, such as `oracle.jdbc.OracleBlob`, `oracle.jdbc.OracleClob`, and so on.

33.1.2 About Using LONG and LONG RAW columns with Application Continuity

Typically, Oracle recommends using LOB columns instead of LONG and LONG RAW columns. This also applies to Application Continuity.

If you have one or more LONG or LONG RAW columns in your table, then the JDBC driver transfers those to the client in streaming mode, when a query selects those columns. In streaming mode, the JDBC driver does not read the column data from the network for LONG or LONG RAW columns, until required. As a result of this, a query involving one or more such columns can result in a replay failure. So, if there is a situation when you cannot avoid using such columns, then perform the following tasks to resolve the issue:

> **See Also:**
>
> About Streaming LONG or LONG RAW Columns

- Use the `defineColumnType` method to redefine the type of the LONG or LONG RAW column. For example, if you redefine the LONG or LONG RAW column as VARCHAR or VARBINARY type, then the driver will not stream the data automatically.
- Declare the types of all the columns involved in the query.

  > **Note:**
  >
  > If you redefine the column types with the `defineColumnType` method, then you must declare the types of all the columns involved in the query. If you do not do so, then the `executeQuery` method will fail.

- Cast the Statement object to `oracle.jdbc.OracleStatement`.

33.2 About Configuring Oracle Database for Application Continuity for Java

You must configure Oracle Database for using Application Continuity for Java.

For using Application Continuity, you must have the following configuration:

- Use Oracle Database 12*c* Release 1 (12.1) or later
- If you are using Oracle Real Application Clusters (Oracle RAC) or Oracle Data Guard, then ensure that FAN is configured with Oracle Notification System (ONS) to communicate with Oracle WebLogic Server or the Universal Connection Pool (UCP)
- Use an application service for all database work. To create the service you must:
  - Run the `SRVCTL` command if you are using Oracle RAC
  - Use the `DBMS_SERVICE` package if you are not using Oracle RAC
- Set the required properties on the service for replay and load balancing. For example, set:
  - `aq_ha_notifications = TRUE` for enabling FAN notification
  - `FAILOVER_TYPE = TRANSACTION` or `FAILOVER_TYPE = AUTO` for using Application Continuity
  - `COMMIT_OUTCOME` = `TRUE` for enabling Transaction Guard
  - `REPLAY_INITIATION_TIMEOUT = 900` for setting the duration in seconds for which replay will occur
  - `FAILOVER_RETRIES = 30` for specifying the number of connection retries for each replay
  - `FAILOVER_DELAY = 10` for specifying the delay in seconds between connection retries
  - `GOAL = SERVICE_TIME`, if you are using Oracle RAC, then this is a recommended setting
  - `CLB_GOAL = LONG`, typically useful for closed workloads. If you are using Oracle RAC, then this is a recommended setting. For most of the other workloads, `SHORT` is the recommended setting.
- Do not use the database service, that is, the default service corresponding to the `DB_NAME` or `DB_UNIQUE_NAME`. This service is reserved for Oracle Enterprise Manager and for DBAs. Oracle does not recommend the use of the database service for high availability because this service cannot be:
  - Enabled and disabled
  - Relocated on Oracle RAC
  - Switched over to Oracle Data Guard

See Also:

- Configuration Examples Related to Application Continuity for Java
- Configuring Oracle Database for Application Continuity

33.3 Application Continuity with DRCP

Oracle Database Release 23ai JDBC driver supports Application Continuity when Database Resident Connection Pooling (DRCP) is enabled on the server side.

For using Application Continuity with DRCP, you must configure an application service to a server that uses DRCP. The following code snippet shows how to use Application Continuity with DRCP:

Example 33-1 Using Application Continuity with DRCP

```
...
String url = "jdbc:oracle:thin:@(DESCRIPTION =
                          (TRANSPORT_CONNECT_TIMEOUT=3000)
                          (RETRY_COUNT=20)(RETRY_DELAY=3s)(FAILOVER=ON)
                          (ADDRESS_LIST =(ADDRESS=(PROTOCOL=tcp)
                          (HOST=CLOUD-SCANVIP.example.com)(PORT=5221))
                          (CONNECT_DATA=(SERVICE_NAME=ac-service)
(SERVER=POOLED)))";
PoolDataSource pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");

// Set DataSource Property
pds.setUser("HR");
pds.setPassword("hr");
System.out.println ("Connecting to " + url);
pds.setURL(url);
pds.setConnectionPoolName("HR-Pool1");
pds.setMinPoolSize(2);
pds.setMaxPoolSize(3);
pds.setInitialPoolSize(2);
Properties prop = new Properties();
prop.put("oracle.jdbc.DRCPConnectionClass", "HR-Pool1");
pds.setConnectionProperties(prop);
...
```

Related Topics

- Overview of Database Resident Connection Pooling

Related Topics

- About Configuring Oracle Database for Application Continuity for Java
  You must configure Oracle Database for using Application Continuity for Java.

33.4 Application Continuity Support for XA Data Source

Oracle Database 12*c* Release 2 (12.2.0.1) introduced a new feature that enhances Application Continuity with support for Oracle XA data source (`javax.sql.XADataSource`), which is similar to non-XA data source (`javax.sql.DataSource`). Both JDBC and Java Transaction API (JTA) allow a JDBC connection to interchangeably participate in local and global/XA transactions.

However, many customer applications obtain connections from an XA data source, but use these connections to perform only local transactions. With the new feature, Application Continuity also covers applications that are using XA-capable data sources but with local

transactions, including local transactions that are promotable to global/XA transactions. So, the benefits of Application Continuity, such as, failover and forward-recovery are extended to these applications.

> **Note:**
>
> You must use Transaction Guard 12.2 or later for using this feature.

Whenever an underlying physical connection participates in a global/XA transaction, or engages in any XA operation, replay is disabled on that connection. All other XA operations function normally, but the application does not get Application Continuity protection.

Once replay is disabled on a connection for the above reasons, it remains disabled until the next request begins. Switching from global/XA transaction to local transaction mode does not automatically re-enable replay on a connection.

To use the XA data source with the local transaction feature, you must specify an Oracle JDBC driver XA data source, similar to the regular Application Continuity configuration (non-XA). The XA data source also depends on the Oracle JDBC driver version that you use, so:

- If you use Oracle JDBC driver 23ai, then you should use `oracle.jdbc.replay.OracleXADataSourceImpl`, although the older `oracle.jdbc.xa.client.OracleXADataSource` also works.
- If you use Oracle JDBC driver 19c or older, then you must use `oracle.jdbc.replay.OracleXADataSourceImpl`.

The following code snippet illustrates how replay is supported for local transactions, and disabled for XA transaction, when the connections are obtained from `OracleXADataSource`:

```
import javax.transaction.xa.*;
import oracle.jdbc.replay.OracleXADataSource;
import oracle.jdbc.replay.OracleXADataSourceFactory;
import oracle.jdbc.replay.ReplayableConnection;

OracleXADataSource xards = OracleXADataSourceFactory.getOracleXADataSource();
xards.setURL(connectURL);
xards.setUser(<user_name>);
xards.setPassword(<password>);
XAConnection xaconn = xards.getXAConnection();

// Implicit request begins
Connection conn = xaconn.getConnection();


/* Local transaction case */
// Request-boundary detection OFF
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
PreparedStatement pstmt=conn.prepareStatement("select
cust_first_name,cust_last_name from customers where customer_id=1");
ResultSet rs=pstmt.executeQuery();

// Outage happens at this point
// Replay happens at this point
rs.next();
```

```
rs.close();
pstmt.close();
((ReplayableConnection) conn).endRequest();
...

/* Global/XA transaction case */
((ReplayableConnection) conn).beginRequest();
conn.setAutoCommit(false);
XAResource xares = xaconn.getXAResource();
Xid xid = createXid();

// Replay is disabled here
xares.start(xid, XAResource.TMNOFLAGS);
conn.prepareStatement("INSERT INTO TEST_TAB VALUES(200, 'another new
record')");

// outage happens at this point
try {

//No replay here and throws exception
conn.executeUpdate();
}

// sqlrexc.getNextException() shows the reason for the replay failure
catch (SQLRecoverableException sqlrexc) {

    …...
}
```

33.5 About Identifying Request Boundaries in Application Continuity for Java

A Request is a unit of work on a physical connection to Oracle Database that is protected by Application Continuity. Request demarcation varies with specific use-case scenarios. A request begins when a connection is borrowed from the Universal Connection Pool (UCP) or WebLogic Server connection pool, and ends when this connection is returned to the connection pool.

The JDBC driver provides explicit request boundary declaration APIs `beginRequest` and `endRequest` in the `oracle.jdbc.OracleConnection` interface. These APIs enable applications, frameworks, and connection pools to indicate to the JDBC Replay Data Source about demarcation points, where it is safe to release the call history, and to enable replay if it had been disabled by a prior request. At the end of the request, the JDBC Replay Data Source purges the recorded history on the connection, where the API is called. This helps to further conserve memory consumption for applications that use the same connections for an extended period of time without returning them to the pool.

For the connection pool to work, the application must get connections when needed, and release connections when not in use. This scales better and provides request boundaries transparently. The APIs have no impact on the applications other than improving resource consumption, recovery, and load balancing performance. These APIs do not involve altering a connection state by calling any JDBC method, SQL, or PL/SQL. An error is returned if an attempt is made to begin or end a request while a local transaction is open.

33.6 Support for Transparent Application Continuity

Oracle Database Release 18c introduced the Transparent Application Continuity feature, which is a functional mode of Application Continuity. Transparent Application Continuity transparently tracks and records session and transactional state, so that a database session can be recovered following recoverable outages. This is performed safely and without the need for any knowledge of the application or application code changes. Transparency is achieved by using a state-tracking infrastructure that categorizes session state usage as an application issues user calls. This feature enables the driver to detect and inject possible request boundaries, which are known as implicit request boundaries. For an implicit request boundary:

- No objects are open
- Cursors are returned to the driver statement cache
- No transactions are open

The session state in such a case is known to be restorable. The driver either closes the current capture and starts a new event, or enables capture if there had been a disabling event. On the next call to the server, the server verifies and, if applicable, creates a request boundary, where there was previously no explicit boundary.

To use Transparent Application Continuity, you must set the server-side service attribute `FAILOVER_TYPE` on the database service to `AUTO`.

Support for implicit request helps to reduce application failover recovery time and optimizes Application Continuity. Using Transparent Application Continuity, the server and the drivers can track transaction and session state usage. However, this feature should be used with caution for applications that change server session states during a request. The JDBC Thin driver provides the `oracle.jdbc.enableImplicitRequests` property to turn off implicit requests, if needed. This property can be set at the system level, which applies to all connections, or at the connection level, which applies to a particular connection. By default, the value of this property is `true`, which means that support for implicit request is enabled.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide*

Starting from Oracle Database Release 19c, if you set the value of the `FAILOVER_TYPE` service attribute to `AUTO`, then the Oracle JDBC driver implicitly begins a request on each new physical connection created through the replay data source. With this feature, applications using third-party connection pools can use Transparent Application Continuity (TAC) easily, without making any code change to inject request boundaries.

This implicit `beginRequest` applies only to the replay data source, and only to the physical connections created during runtime, when TAC is enabled. The driver does not implicitly begin a request after each reconnection during replay attempts (that is, a `beginRequest` is not implicitly injected during a replay connection) or in manual Application Continuity mode, where the `FAILOVER_TYPE` service attribute is set to `TRANSACTION`.

If you want to turn off this feature explicitly, you can set the value of the Java system property `oracle.jdbc.beginRequestAtConnectionCreation` to `false`. The default value of this property is `true`.

Related Topics

- About Enabling FAILOVER_RESTORE
  This section describes how to use the `FAILOVER_RESTORE` service attribute.

33.6.1 Support for Resumable Cursors

Starting from Release 23ai, Oracle JDBC drivers support resumable cursors by default, when using Transparent Application Continuity (TAC). Typically, these cursors are long-running cursors that stay open outside of transactional work. This support increases TAC protection coverage for applications with resumable cursors because it establishes implicit request boundaries more often, even when such cursors are open.

A resumable cursor can remain valid for the entire duration of an explicit request, and can be replayed separately from the main request. Such cursors are commonly used in batch processes. A cursor is considered as a resumable cursor if it meets the following criteria:

- The cursor must be opened when connected to a TAC service
- The cursor must be open when:
  - The session has no open transaction in progress
  - The non-transactional session state is either client restorable or template restorable
- It must be possible to re-execute the query using the system change number (SCN), if replay is required
- The cursor must not create, modify, or read session state during a fetch call, if the fetch call occurs during a round trip that is separate from the cursor execution call

Basically, session state referenced by the cursor must be frozen at the conclusion of the execution phase. Some examples of cursors that are generally not classified as resumable cursors are the cursors that reference the following:

- `sequence.nextval`, `sequence.currval`
- `sys_context` or application context
- PL/SQL functions

> **Note:**
>
> The preceding references are still disqualifying if they occur during fetch-time as a result of VPD-imposed clauses.

> **See Also:**
>
> *Real Application Clusters Administration and Deployment Guide*

33.7 Establishing the Initial State Before Application Continuity Replays

Non-transactional session state (NTSS) is state of a database session that exists outside database transactions and is not protected by recovery. For applications that use stateful requests, the non-transactional state is re-established as the rebuilt session.

For applications that set state only at the beginning of a request, or for stateful applications that gain performance benefits from using connections with a preset state, one among the following callback options are provided:

- No Callback
- Connection Labeling
- Connection Initialization Callback
- About Enabling FAILOVER_RESTORE

33.7.1 No Callback

In this scenario, the application builds up its own state during each request.

33.7.2 Connection Labeling

This scenario is applicable only to Universal Connection Pool (UCP) and Oracle WebLogic Server. You can modify the application to take advantage of the preset state on connections. Connection Labeling APIs first determine how well a connection matches, and then use a callback to populate the gap when a connection is borrowed.

> **Note:**
>
> If you are using connection labeling, then you cannot set the `RESET_STATE` service attribute to `LEVEL1` or `LEVEL2`.

> **See Also:**
>
> *Oracle Universal Connection Pool Developer's Guide*

33.7.3 Connection Initialization Callback

In this scenario, the replay data source uses an application callback to set the initial state of the session during runtime and replay. The JDBC Replay Data Source provides an optional connection initialization callback interface as well as methods for registering and unregistering such callbacks.

When registered, the initialization callback is executed at each successful reconnection following a recoverable error. An application is responsible for ensuring that the initialization actions are the same as that on the original connection before failover. If the callback invocation fails, then replay is disabled on that connection.

This section discusses initialization callbacks in the following sections:

- Creating an Initialization Callback
- Registering an Initialization Callback
- Removing or Unregistering an Initialization Callback

33.7.3.1 Creating an Initialization Callback

To create a JDBC connection initialization callback, an application implements the `oracle.jdbc.replay.ConnectionInitializationCallback` interface. One callback is allowed for every instance of the `oracle.jdbc.replay.OracleDataSource` interface.

> **Note:**
>
> This callback is only invoked during failover, after a successful reconnection.

Example

The following code snippet demonstrates a simple initialization callback implementation:

```
import oracle.jdbc.replay.ConnectionInitializationCallback;
class MyConnectionInitializationCallback implements ConnectionInitializationCallback
{
    public MyConnectionInitializationCallback()
    {
        ...
    }
    public void initialize(java.sql.Connection connection) throws SQLException
    {
        // Reset the state for the connection, if necessary (like ALTER SESSION)
        ...
    }
 }
```

For applications using an XA data source, the connection initialization callback is registered on the XA replay data source. The callback is executed every time when *both* of the following happen:

- A connection is borrowed from the connection pool.
- The replay XA data source gets a new physical connection at failover.

> **Note:**
>
> The connection initialization must be idempotent. If the connection is already initialized, then it must not repeat itself. This enables applications to reestablish session initial starting point after a failover and before the starting of replay. The callback execution must leave an open local transaction without committing it or rolling it back. If this is violated, an exception is thrown.

If a callback invocation fails, replay is disabled on that connection. For example, an application embeds the set up phase for a connection in this callback.

33.7.3.2 Registering an Initialization Callback

Use the following method that the JDBC Replay Data Source provides in the `oracle.jdbc.replay.OracleDataSource` interface for registering a connection initialization callback:

```
registerConnectionInitializationCallback(ConnectionInitializationCallback cbk)
```

One callback is allowed for every instance of the `OracleDataSource` interface.

For using an XA Data Source, use the `registerConnectionInitializationCallback(ConnectionInitializationCallback cbk)` method in the `oracle.jdbc.replay.OracleXADataSource` interface.

33.7.3.3 Removing or Unregistering an Initialization Callback

Use the following method that the JDBC Replay Data Source provides in the `oracle.jdbc.replay.OracleDataSource` interface for unregistering a connection initialization callback:

```
unregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)
```

For using an XA Data Source, use the `unregisterConnectionInitializationCallback(ConnectionInitializationCallback cbk)` method in the `oracle.jdbc.replay.OracleXADataSource` interface.

33.7.4 About Enabling FAILOVER_RESTORE

This section describes how to use the `FAILOVER_RESTORE` service attribute.

`FAILOVER_RESTORE` service attribute was introduced in Oracle Database 12*c* Release 2 (12.2.0.1). Setting the `FAILOVER_RESTORE` attribute to `LEVEL1` automatically restores the common initial state before replaying a request. By default, the value of the `FAILOVER_RESTORE` attribute is set to `NONE`, which means that it is disabled.

Starting from Oracle Database Release 18c, you can also set the value of this attribute to `AUTO`. Also, if you set the value of the `FAILOVER_TYPE` attribute to `AUTO`, then `FAILOVER_RESTORE` is set to `AUTO` automatically. You cannot change the value of `FAILOVER_RESTORE` to anything else as long as `FAILOVER_TYPE` is set to `AUTO`. When `FAILOVER_RESTORE` is set to `AUTO`, then the common initial state is also set. As far as session state restore is concerned, this setting provides the same function as `FAILOVER_RESTORE` set to `LEVEL1`.

> **See Also:**
>
> *Real Application Clusters Administration and Deployment Guide*

Note:

For Application Continuity for Java available with Oracle Database Release 23ai, the following initial states are supported for `FAILOVER_RESTORE`:

- `NLS_CALENDAR`
- `NLS_CURRENCY`
- `NLS_DATE_FORMAT`
- `NLS_DATE_LANGUAGE`
- `NLS_DUAL_CURRENCY`
- `NLS_ISO_CURRENCY`
- `NLS_LANGUAGE`
- `NLS_LENGTH_SEMANTICS`
- `NLS_NCHAR_CONV_EXCP`
- `NLS_NUMERIC_CHARACTER`
- `NLS_SORT`
- `NLS_TERRITORY`
- `NLS_TIME_FORMAT`
- `NLS_TIME_TZ_FORMAT`
- `NLS_TIMESTAMP_FORMAT`
- `NLS_TIMESTAMP_TZ_FORMAT`
- `TIME_ZONE (OCI, ODP.NET 12201)`
- `CURRENT_SCHEMA`
- `MODULE`
- `ACTION`
- `CLIENT_ID`
- `ECONTEXT_ID`
- `ECONTEXT_SEQ`
- `DB_OP`
- `AUTOCOMMIT` states (Java and SQL*Plus)
- `CONTAINER` (PDB) and `SERVICE` for OCI and ODP.NET

Starting from Oracle Database Release 19c, the following additional initial states are supported for `FAILOVER_RESTORE`:

- `ERROR_ON_OVERLAP_TIME`
- `EDITION`
- `SQL_TRANSLATION_PROFILE`
- `ROW ARCHIVAL VISIBILITY`
- `ROLEs`

- CLIENT_INFO

Note:

Since Oracle Database Release 19c, FAILOVER_RESTORE is automatically enabled in Transparent Application Continuity (TAC) mode.

The following states are excluded from the auto-restoration option because they are not supported by the Thin driver:

- NLS_COMP
- CALL_COLLECT_TIME

For many applications, enabling FAILOVER_RESTORE is sufficient to automatically restore the initial state required for AC replay, without the use of a callback. If your application requires any initial state that is not mentioned in the preceding list, or if the application prefers explicit control over setting the initial state, then the application must use a callback, either connection labeling or an initialization callback. When a callback is configured, it overrides the initial states restored by FAILOVER_RESTORE, in case the latter is enabled at the same time.

33.8 About Delaying the Reconnection in Application Continuity for Java

By default, when JDBC Replay Data Source initiates a failover, the driver attempts to recover the in-flight work at an instance where the service is available. For doing this, the driver must first reestablish a good connection to a working instance. This reconnection can take some time if the database or the instance needs to be restarted before the service is relocated and published. So, the failover should be delayed until the service is available from another instance or database.

You must use the FAILOVER_RETRIES and FAILOVER_DELAY parameters to sustain the delay because maximum delay is calculated as FAILOVER_RETRIES multiplied by FAILOVER_DELAY. These parameters can work well in conjunction with a planned outage, for example, an outage that may make a service unavailable for several minutes. While setting the FAILOVER_DELAY and FAILOVER_RETRIES parameters, check the value of the REPLAY_INITIAITION_TIMEOUT parameter first. The default value for this parameter is 900 seconds. A high value for the FAILOVER_DELAY parameter can cause replay to be canceled.

Parameter Name	Possible Value	Default Value
FAILOVER_RETRIES	Positive integer zero or above	30
FAILOVER_DELAY	Time in seconds	10

33.8.1 Configuration Examples Related to Application Continuity for Java

This section provides configuration examples for service creation and modification in the following subsections:

- Creating Services on Oracle RAC

- Modifying Services on Single-Instance Databases

33.8.1.1 Creating Services on Oracle RAC

If you are using Oracle RAC or Oracle RAC One, then use the `SRVCTL` command to create and modify services in the following way:

For Transparent Application Continuity

You can create services that use Transparent Application Continuity, as follows:

For policy-managed databases:

```
$ srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -
clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore AUTO -failoverretry 30
-failoverdelay 10 -commit_outcome TRUE -failovertype AUTO -replay_init_time
1800 -retention 86400  -notification TRUE
```

For administrator-managed databases:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -available
serv2 -clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore AUTO
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype AUTO -
replay_init_time 1800 -retention 86400 -notification TRUE
```

For Manual Application Continuity

You can create services that use manual Application Continuity, as follows:

For policy-managed databases:

```
$ srvctl add service -db codedb -service GOLD -serverpool ora.Srvpool -
clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore LEVEL1 -failoverretry
30
-failoverdelay 10 -commit_outcome TRUE -failovertype TRANSACTION -
replay_init_time 1800 -retention 86400 -notification TRUE
```

For administrator-managed databases:

```
$ srvctl add service -db codedb -service GOLD -preferred serv1 -available
serv2 -clbgoal SHORT -rlbgoal SERVICE_TIME -failover_restore LEVEL1
-failoverretry 30 -failoverdelay 10 -commit_outcome TRUE -failovertype
TRANSACTION -replay_init_time 1800 -retention 86400 -notification TRUE
```

33.8.1.2 Modifying Services on Single-Instance Databases

If you are using a single-instance database, then use the `DBMS_SERVICE` package to modify services in the following way:

```
declare
params dbms_service.svc_parameter_array;
begin
params('FAILOVER_TYPE'):='TRANSACTION';
params('REPLAY_INITIATION_TIMEOUT'):=1800;
```

```
params('RETENTION_TIMEOUT'):=604800;
params('FAILOVER_DELAY'):=10;
params('FAILOVER_RETRIES'):=30;
params('commit_outcome'):='true';
params('aq_ha_notifications'):='true';
dbms_service.modify_service('[your service]',params);
end;
/
```

33.9 About Retaining Mutable Values in Application Continuity for Java

A mutable object is a variable, function return value, or other structure that returns a different value each time that it is called. For example, `Sequence.NextVal`, `SYSDATE`, `SYSTIMESTAMP`, and `SYS_GUID`. To retain the function results for named functions at replay, the DBA must grant `KEEP` privileges to the user who invokes the function. This security restriction is imposed to ensure that it is valid for replay to save and restore function results for code that is not owned by that user.

> **See Also:**
>
> *Oracle Database Development Guide*

33.9.1 Grant and Revoke Interface

You can work with mutables values by using the standard `GRANT` and `REVOKE` interfaces in the following way:

- Dates and SYS_GUID Syntax
- Sequence Syntax
- GRANT ALL Statement
- Rules for Grants on Mutable Values

33.9.1.1 Dates and SYS_GUID Syntax

The `DATE_TIME` and `SYS_GUID` syntax is as follows:

```
GRANT [KEEP DATE TIME|SYSGUID]..[to USER}
REVOKE [KEEP DATE TIME | KEEP SYSGUID] … [from USER]
```

For example, for EBS standard usage with original dates

```
Grant KEEP DATE TIME,  KEEP SYSGUID  to [custom user];
Grant KEEP DATE TIME,  KEEP SYSGUID  to [apps user];
```

33.9.1.2 Sequence Syntax

The Sequence syntax can be of the following types:

- Owned Sequence Syntax
- Others Sequence Syntax

Owned Sequence Syntax

```
ALTER SEQUENCE [sequence object] [KEEP|NOKEEP];
```

This command retains the original values of `sequence.nextval` for replaying, so that the keys match after replay. Most applications need to retain the sequence values at replay. The `ALTER SYNTAX` is only for owned sequences.

Others Sequence Syntax

```
GRANT KEEP SEQUENCE..[to USER] on [sequence object];
REVOKE  KEEP SEQUENCE … [from USER]  on [sequence object];
```

For example, use the following command for EBS standard usage with original sequence values:

```
Grant KEEP SEQUENCE  to [apps user] on [sequence object];
Grant KEEP SEQUENCE  to [custom user] on [sequence object];
```

33.9.1.3 GRANT ALL Statement

The `GRANT ALL` statement grants `KEEP` privilege on all the objects of a user. However, it excludes mutable values, that is, mutable values require explicit grants.

33.9.1.4 Rules for Grants on Mutable Values

Follow these rules while granting privileges on mutable objects:

- If a user has `KEEP` privilege granted on mutables values, then the objects inherit mutable access when the `SYS_GUID`, `SYSDATE`, and `SYSTIMESTAMP` functions are called.
- If the `KEEP` privilege on mutable values on a sequence object is revoked, then SQL or PL/SQL blocks using that object will not allow mutable collection or application for that sequence.
- If granted privileges are revoked between runtime and failover, then the mutable values that are collected are not applied for replay.
- If new privileges are granted between runtime and failover, mutable values are not collected and these values are not applied for replay.

33.10 Application Continuity Statistics

The JDBC Replay Data Source supports the following statistics for an application using Application Continuity:

- Total number of requests
- Total number of completed requests
- Total number of calls
- Total number of protected calls
- Total number of calls affected by outages
- Total number of calls triggering replay
- Total number of calls affected by outages during replay

- Total number of successful replay
- Total number of failed replay
- Total number of disabled replay
- Total number of replay attempts

All these metrics are available both on a per-connection basis and across-connections basis. You can use the following methods for obtaining these statistics:

- `getReplayStatistics(StatisticsReportType)`

  Use the `oracle.jdbc.replay.ReplayableConnection.getReplayStatistics(StatisticsReportType)` method to obtain the snapshot statistics. The argument to this method is an enum type also defined in the same `ReplayableConnection` interface. To obtain statistics across connections, it is best calling this method after the main application logic. Applications can either use any `oracle.jdbc.replay.ReplayableConnection` that is still open, or open a new connection to the same data source. This applies to applications using both UCP and WLS data sources, and applications that directly use the replay data source.

- `getReplayStatistics()`

  Use the `oracle.jdbc.replay.OracleDataSource.getReplayStatistics()` method to obtain across-connection statistics. This applies only to applications that directly use replay data source.

Both methods return an `oracle.jdbc.replay.ReplayStatistics` object, from which you can retrieve individual replay metrics. The following is a sample output that prints a ReplayStatistics object as String:

```
AC Statistics:
=============================================
TotalRequests = 1
TotalCompletedRequests = 1
TotalCalls = 19
TotalProtectedCalls = 19
=============================================
TotalCallsAffectedByOutages = 3
TotalCallsTriggeringReplay = 3
TotalCallsAffectedByOutagesDuringReplay = 0
=============================================
SuccessfulReplayCount = 3
FailedReplayCount = 0
ReplayDisablingCount = 0
TotalReplayAttempts = 3
=============================================
```

If you want to clear the accumulated replay statistics per connection or for all connections, then you can use the following methods:

- `oracle.jdbc.replay.ReplayableConnection.clearReplayStatistics(ReplayableConnection.StatisticsReportType reportType)`
- `oracle.jdbc.replay.OracleDataSource.clearReplayStatistics()`

> **Note:**
>
> All statistics reflect only updates since the latest clearing.

33.11 About Disabling Replay in Application Continuity for Java

This section describes the following concepts:

- How to Disable Replay
- When to Disable Replay
- Diagnostics and Tracing

33.11.1 How to Disable Replay

If any application module uses a design that is unsuitable for replay, then the disable replay API disables replay on a per request basis. Disabling replay can be added to the callback or to the main code by using the `disableReplay` method of the `oracle.jdbc.replay.ReplayableConnection` interface. For example:

```
if (connection instanceof oracle.jdbc.replay.ReplayableConnection)
{
    (( oracle.jdbc.replay.ReplayableConnection)connection).disableReplay();

}
```

Disabling replay does not alter the connection state by reexecuting any JDBC method, SQL or PL/SQL. When replay is disabled using the disable replay API, both recording and replay are disabled until that request ends. There is no API to reenable replay because it is invalid to reestablish the database session with time gaps in a replayed request. This ensures that replay runs *only* if a complete history of needed calls has been recorded.

33.11.2 When to Disable Replay

By default, the JDBC Replay Data Source replays following a recoverable error. The disable replay API can be used in the entry point of application modules that are unable to lose the database sessions and recover. For example, if the application uses the `UTL_SMTP` package and does not want messages to be repeated, then the `disableReplay` API affects only the request that needs to be disabled. All other requests continue to be replayed.

The following are scenarios to consider before configuring an application for replay:

- Application Calls External PL/SQL Actions that Should not Be Repeated
- Application Synchronizes Independent Sessions
- Application Uses Time at the Middle-tier in the Execution Logic
- Application assumes that ROWIds do not change
- Application Assumes that Side Effects Execute Once
- Application Assumes that Location Values Do not Change

33.11.2.1 Application Calls External Systems that Should not Be Repeated

During replay, autonomous transactions and external systems (like PL/SQL calls or Java calls) can have side effects that are separate from the main transaction. These side effects are replayed unless you specify otherwise and leave persistent results behind. These side effects include writing to an external table, sending email, forking sessions out of PL/SQL or Java, transferring files, accessing external URLs, and so on. For example, in case of PL/SQL messaging, suppose, you walk away in-between some work without committing and the session times out. Now, if you issue a `Ctrl+C` command, then the foreground of a component fails. When you resubmit the work, then this side effect can also be repeated.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about potential side effects of Application Continuity

You must make a conscious decision about whether to enable replay for external actions or not. For example, you can consider the following situations where this decision is important:

- Using the `UTL_HTTP` package to issue a SOA call
- Using the `UTL_SMTP` package to send a message
- Using the `UTL_URL` package to access a web site

Use the `disableReplay` API if you do not want such external actions to be replayed.

33.11.2.2 Application Synchronizes Independent Sessions

You can configure an application for replay if the application synchronizes independent sessions using volatile entities that are held until commit, rollback, or session loss. In this case, the application synchronizes multiple sessions connected to several data sources that are otherwise inter-dependent using resources such as a database lock. This synchronization may be fine if the application is only serializing these sessions and understands that any session may fail. However, if the application assumes that a lock or any other volatile resource held by one data source implies exclusive access to data on the same or a separate data source from other connections, then this assumption may be invalidated when replaying.

During replay, the driver is not aware that the sessions are dependent on one session holding a lock or other volatile resource. You can also use pipes, buffered queues, stored procedures taking a resource (such as a semaphore, device, or socket) to implement the synchronization that are lost by failures.

> **Note:**
>
> The `DBMS_LOCK` does not replay in the restricted version.

33.11.2.3 Application Uses Time at the Middle-tier in the Execution Logic

In this case, the application uses the wall clock at the middle-tier as part of the execution logic. The JDBC Replay Data Source does not repeat the middle-tier time logic, but uses the

database calls that execute as part of this logic. For example, an application using middle-tier time may assume that a statement executed at Time T1 is not reexecuted at Time T2, unless the application explicitly does so.

33.11.2.4 Application assumes that ROWIds do not change

If an application caches ROWIDs, then access to these ROWIDs may be invalidated due to database changes. Although a ROWID uniquely identifies a row in a table, a ROWID may change its value in the following situations:

- The underlying table is reorganized
- An index is created on the table
- The underlying table is partitioned
- The underlying table is migrated
- The underlying table is exported and imported using EXP/IMP/DUL
- The underlying table is rebuilt using Golden Gate or Logical Standby or other replication technology
- The database of the underlying table is flashed back or restored

It is bad practice for an application to store ROWIDs for later use as the corresponding row may either not exist or contain completely different data.

33.11.2.5 Application Assumes that Side Effects Execute Once

In this case, the following are replayed during a replay:

- Autonomous transactions
- Opening of back channels separate to the main transaction side effects

Examples of back channels separate to the main transaction include writing to an external table, sending email, forking sessions out of PL/SQL or Java, writing to output files, transferring files, and writing exception files. Any of these actions leave persistent side effects in the absence of replay. Back channels can leave persistent results behind. For example, if a user leaves a transaction midway without committing and the session times out, then the user presses Ctrl+C, the foreground or any component fails. If the user resubmits work, then the side effects can be repeated.

33.11.2.6 Application Assumes that Location Values Do not Change

`SYSCONTEXT` options comprise a location-independent set such as National Language Support (NLS) settings, `ISDBA`, `CLIENT_IDENTIFIER`, `MODULE`, and `ACTION`, and a location-dependent set that uses physical locators. Typically, an application does not use the physical identifier, except in testing environments. If physical locators are used in mainline code, then the replay finds the mismatch and rejects it. However, it is fine to use physical locators in callbacks.

Example

```
select
    sys_context('USERENV','DB_NAME')
    ,sys_context('USERENV','HOST')
    ,sys_context('USERENV','INSTANCE')
    ,sys_context('USERENV','IP_ADDRESS')
    ,sys_context('USERENV','ISDBA')
    ,sys_context('USERENV','SESSIONID')
    ,sys_context('USERENV','TERMINAL')
```

```
    ,sys_context('USERENV',ID')
from dual
```

33.11.3 Diagnostics and Tracing

The JDBC Replay Data Source supports standard JDK logging. Logging is enabled using the Java command-line `-Djava.util.logging.config.file=<file>` option. Log level is controlled with the `oracle.jdbc.level` attribute in the log configuration file. For example:

```
oracle.jdbc.level = FINER|FINEST
```

where, `FINER` produces external APIs and `FINEST` produces large volumes of trace. You must use `FINEST` only under supervision.

If you use the `java.util.logging.XMLFormatter` class to format a log record, then the logs are more readable but larger. If you are using replay with FAN enabled on UCP or WebLogic Server, then you should also enable FAN-processing logging.

> **See Also:**
>
> - Diagnosability in JDBC
> - *Oracle Universal Connection Pool for JDBC Developer's Guide*

33.11.3.1 Writing Replay Trace to Console

Following is the example of a configuration file for logging configuration.

```
oracle.jdbc.level = FINER
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.XMLFormatter
```

33.11.3.2 Writing Replay Trace to a File

Following is the example of a `properties` file for logging configuration.

```
oracle.jdbc.level = FINEST

# Output File Format (size, number and style)
# count: Number of output files to cycle through, by appending an integer to the base
file name:
# limit: Limiting size of output file in bytes
handlers = java.util.logging.FileHandler
java.util.logging.FileHandler.pattern = [file location]/replay_%U.trc
java.util.logging.FileHandler.limit = 500000000
java.util.logging.FileHandler.count = 1000
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter
```

> **See Also:**
>
> Restrictions and Other Considerations for Application Continuity