ORACLE
Autonomous
Database Cloud

Java Programming with Oracle Database 19c

On-Premises, Cloud, Data Types, Security, Performance/Scalability, Zero Downtime

WHITE PAPER / JANUARY 9, 2020

ORACLE

DISCLAIMER

INTRODUCTION

The Oracle database release 19c enables Java developers and architects to design and deploy modern, secure, fast and resilient applications using the Oracle JDBC drivers, the Universal Connection Pool (UCP), and the database-embedded JVM (a.k.a. OJVM)[1].

This white paper walks you through the latest enhancements and APIs in the following areas: JDBC standards, connectivity to database on-premises and/or in the Cloud, new data types support, security enhancements, performance and scalability, and zero downtime.

JDBC STANDARDS

The Oracle JDBC supports the standard JDBC 4.3 enhancements.

JAVA CONNECTIVITY TO DB 19C ON-PREMISES & IN THE CLOUD

This section covers: database alias, database service, the Easy Connect Naming mechanism and connectivity to databases in the Cloud.

Drivers Jars

In this release, the core Oracle JDBC driver comes as:

- `ojdbc8.jar` compiled w Java SE 8 (JDBC 4.2); can be used with Java 11
- `ojdbc10.jar` compiled w Java SE 10 (JDBC 4.3); can be used with Java 11

The Metalink Doc ID 2482279.1 gives a statement of direction for Oracle JDBC releases.
The complete list of all the jars files that you might need can be found on Oracle Maven as well as the OTN download page.

Database alias and service

For connecting to an Oracle database, Java programs use an Oracle Net Naming alias in the JDBC connect string e.g., `jdbc:oracle:thin:@dbalias`
The Oracle Net Services alias is expanded into a full description that includes: the protocol, the host, the port and the service name. A configuration file known as `tnsnames.ora` or an LDAP a directory naming service repository `ldap.ora` (for large scale deployments). The Java developer or the database administrator define the mapping of the alias and the full description in the configuration file or the directory naming service repository.
Here is an example of `tnsnames.ora` entry

```
dbalias =
     (DESCRIPTION=
        (ADDRESS=
           (PROTOCOL=tcp)
           (HOST=sales-server)
           (PORT=1521))
```

[1] See the OJVM landing page for more details (including Github code samples) @ https://www.oracle.com/database/technologies/appdev/ojvm.html

```
         (CONNECT_DATA= (SERVICE_NAME=dbservice)))
```

JDBC supports the long URL format where the full description of the `tnsnames.ora` entry is specified directly in the connect string, thereby avoiding the use of the configuration file or a directory naming service.

For example

```
jdbc:oracle:thin:@ DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=sales-
server) (PORT=1521))(CONNECT_DATA= (SERVICE_NAME=dbservice)))
```

The database service name is similar to JNDI names for datasources, the database service name virtualizes the database; the associated database may be changed e.g., service relocation from one instance to another in RAC environments or moved from on-premises to the Cloud without making code change.
The following description is the recommended database connect string which supports clustered database (i.e., RAC) and disaster recovery (i.e., Active Data Gard) environments.

```
dbalias =
       (DESCRIPTION =
          (CONNECT_TIMEOUT=120)(RETRY_COUNT=20)(RETRY_DELAY=3)
          (TRANSPORT_CONNECT_TIMEOUT=3)
          (ADDRESS_LIST =
             (LOAD_BALANCE=on)
             (ADDRESS = (PROTOCOL = TCP)(HOST=primary-scan)(PORT=1521)))
           (ADDRESS_LIST =
             (LOAD_BALANCE=on)
             (ADDRESS = (PROTOCOL = TCP)(HOST=standby-scan )(PORT=1521)))
          (CONNECT_DATA=(SERVICE_NAME = service name)))
```

The specification above has provision for retrying to connect, load-balancing the workload across all the instance accessing the same database (RAC technology) and failing over to another data center in the case of disaster recovery (Active Data Guard technology).

THE LOCATION OF THE CONFIGURATION FILE

The default location for the `tnsnames.ora` configuration file is specified by `TNS_ADMIN` specified as system property: `oracle.net.tns_admin`. Starting with Oracle database release 18c, JDBC supports defining TNS_ADMIN as environment variable or in the URL, as follows:

```
jdbc:oracle:thin:@//myhost:1521/orcl?TNS_ADMIN=/home/oracle/network/
admin/
```

Properties in JDBC Connect String

Starting with this release, the JDBC properties can be set in the connect string. In the following example, the implicit statement cache size is set to value 60.

```
jdbc:oracle:thin:@(description=
 (address=(protocol=tcps)(port=1521)(host=example1.com))
 (connect_data=(service_name=example2.com)))?
oracle.jdbc.implicitStatementCacheSize=60
```

Properties Files

Starting with Oracle database 18c, the JDBC drivers support a properties file mechanism for simplifying Cloud as well as on-premises connectivity. The default properties file is `ojdbc.properties`, and it's default location is defined by the value of TNS_ADMIN (value set

either in the JDBC URL or via a system property (`oracle.net.tns_admin`) or environment variable)
i.e., `$TNS_ADMIN/ojdbc.properties`.

You may use an additional file, named from your database alias ojdbc_<dbalias>_properties (e.g., `ojdbc_orcl_properties`). If both the default and the non-default files are present, then the non-default file takes precedence. The online JDBC Javadoc, gives more details on the properties file.

Wallet Location Property

Starting with Oracle Database release 18c, the JDBC driver supports a new property `my_wallet_directory` for specifying the location of the wallets as follows:

```
dbaccess =
   (DESCRIPTION=
      (ADDRESS=
        (PROTOCOL=tcps)
        (Host=hostname)
        (Port=1522))
    (CONNECT_DATA=
         (SERVICE_NAME=myservicename))
         (Security=(my_wallet_directory=$TNS_ADMIN/jnetadmin_c/)))
```

You may also set the wallet location via the `oracle.net.wallet_location` system property, as follows:

```
java -cp .:oraclepki.jar:ojdbc10.jar -D oracle.net.wallet_location=
file:/ path/to/wallet/cwallet.sso MyApp
```

Easy Connect Plus

The Easy Connect Naming mechanism available since Oracle database release 11, dynamically expands the connect string without the need for an alias (and thereby no need for a configuration or directory naming service), using default values for the port, the protocol, and the service; the default service name however, needs to be defined in the `DEFAULT_SERVICE_listener.ora` file.
The general Easy Connect syntax is
`[//]host[:port][/service_name][:server][/instance_name]`

The following Java connect string `jdbc:oracle:thin:@sales-server` will expand into `jdbc:oracle:thin@//sales-server:1521`

In this release, the Easy Connect Naming has been enhanced to support: TCPS, multiple hosts or ports, a global database service name, an optional wallet location, an optional distinguished database server name, and passing connections properties as name-value pairs in the connect string.

The general Easy Connect Plus syntax is

```
"[[protocol:]//]host1[,host12,host13][:port1][,host2:port2][/service
_name][:server][/instance_name][?[wallet_location=dir][&ssl_server_c
ert_dn=dn],...]"
```

The question mark (?) indicates the start of name-value pairs and the ampersand (&) is the delimiter between the name-value pairs.

The following Easy Connect Plus based connect string
```
jdbc:oracle:thin:@tcp://salesserver1:1521, salesserver2,
salesserver3:1522/sales.us.example.com
```

will expands into

```
jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=
      (ADDRESS=(PROTOCOL=tcp)(HOST=salesserver1)(PORT=1521))
      (ADDRESS=(PROTOCOL=tcp)(HOST=salesserver2)(PORT=1522))
      (ADDRESS=(PROTOCOL=tcp)(HOST=salesserver3)(PORT=1522)))
       (CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

All JDBC connection properties (including the server's domain name and many others) may be specified in the URL, using the Easy Connect Plus syntax.

For example, the following connect string

```
jdbc:oracle:thin:@tcp://myorclhostname:1521/myorclservicename?
oracle.jdbc.implicitStatementCacheSize=100
```

Will expands into

```
jdbc:oracle:thin:@(description=(address=(protocol=tcp)(port=1521)(ho
st=myorclhostname))(connect_data=(service_name=myorclservicename)))?
oracle.jdbc.implicitStatementCacheSize=100
```

See section 8.2.5 of the JDBC documentation for more details.

Java Connectivity to Databases in the Cloud

Connecting your Java applications to the Autonomous Cloud database services i.e., the Oracle Autonomous Transaction Processing (ATP-S, ATP-D) or the Oracle Autonomous Data Warehouse Cloud (ADW) requires the following simple configurations steps.

- The most important step is to get the client credentials from the Cloud console; it is a zip file containing the configuration files mainly: `tnsnames.ora, sqlnet.ora, ojdbc.properties, keystore.jks, truststore.jks, cwallet.ora and ewallet.p12` see more details here (see step #2, under the prerequisites of this document)
- Get the latest JDK8 (JDK8u163+), JDK9, JDK10 or JDK11
- Get the latest Oracle JDBC and UCP jars from either the Oracle Maven repository or the JDBC download page. Note that 19.3 JDBC drivers are available on central maven.
- The `ojdbc.properties` property file (and `ojdbc_<aliasname>.properties`) is pre-configured to work with Oracle Wallet, out of the box; you need to configure it for JKS, as follows.

```
oracle.net.ssl_server_dn_match=true
javax.net.ssl.trustStore=${TNS_ADMIN}/truststore.jks
javax.net.ssl.trustStorePassword=welcome1
javax.net.ssl.keyStore=${TNS_ADMIN}/keystore.jks ***Not needed for ATP-D
javax.net.ssl.keyStorePassword=welcome1.         ***Not needed for ATP-D
```

This blog post zooms on ATP-D. Visit the Java connectivity to DB Cloud Service page for more details.

DATA TYPES SUPPORT

JSON Datatype Verification

Since Oracle Database 18c, the `isColumnJSON()` method may be used to check that a returned column is of JSON datatype.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);
```

```
ResultSetMetaData rsmd = rs.getMetaData();

OracleResultSetMetaData orsmd = (OracleResultSetMetaData)rsmd;
...
boolean json = orsmd.isColumnJSON(i);
```

REF CURSOR as IN bind Variable

See the GitHub example.

SECURITY ENHANCEMENTS

In the previous Oracle database releases, the Oracle JDBC driver has supported several security mechanisms including: support for strong authentication, data encryption and integrity, SSL, Kerberos, RADIUS, secure external password store, and Oracle Advanced Security.
The latest security capabilities for Java applications includes: server domain name verification, automatic SSL authentication, and HTTP Proxy configuration.

Server Domain Name Verification

Starting with Oracle Database 18c, the JDBC driver automatically authenticate the server if its DN is specified either via `oracle.net.ssl_server_cert_dn` connection property or via `ssl_server_cert_dn` in the JDBC URL.

```
oracle.net.ssl_server_cert_dn="CN=test.us1.oracletest.com,OU=ST,O=Or
acle,ST=California,C=US"
```

Note: the value set in the URL overrides the value set in the property.

Automatic SSL Authentication

AUTOMATIC RESOLUTION OF PUBLIC KEY INFRASTRUCTURE

Oracle provides a public key infrastructure (PKI) for using public keys and certificates however, the Oracle PKI provider must be registered. Starting with Oracle Database release 18c, the JDBC driver resolves the PKI provider automatically (i.e., no need to register it) by loading the OraclePKIProvider, **if** the provider implementation i.e., `oraclepki.jar` is on the CLASSPATH **or if** the `oracle.net.wallet_location` connection property or system property is set. Note that `osdt_core.jar` and `osdt_cert.jar` should be in the `CLASSPATH`. These additional jars are downloaded along with 19.3 JDBC driver on Central Maven or you can download ojdbc8-full.zip or ojdbc10-full.zip to get these jars.

The wallet location can be set in one of the following two formats:

- `file:/path/ewallet.sso" or "file:/path/cwallet.p12" or "file:/path/to/directory/`
- `(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY=/path/to/directory))) -Doracle.net.wallet_location= '(SOURCE=(METHOD=FILE)(METHOD_DATA=(DIRECTORY= ...)))'`

Using Oracle Wallets SSL authentication with Java also requires `osdt_core.jar` and `osdt_cert.jar` in the `CLASSPATH`.

```
java -cp
../ojdbc10.jar:./oraclepki.jar:./osdt_core.jar:./osdt_cert.jar:.
StatementSample
```

AUTOMATIC KEY STORE TYPE RESOLUTION

Staring with the Oracle database 18c, the JDBC driver can resolve the key store types based on the extension of the keystore (the value of the javax.net.ssl.keyStore property) and truststore (the value of the javax.net.ssl.trustStore property) files.
a) The file extension `.jks` resolves to `javax.net.ssl.keyStoreType` as JKS
b) The file extension `.sso` resolves to `javax.net.ssl.keyStoreType` as SSO
c) The file extension `.p12` resolves to `javax.net.ssl.keyStoreType` as PKCS12
d) The file extension `.pfx` resolves to `javax.net.ssl.keyStoreType` as PKCS12

If the key store or the trust store is a URI with a kss:// scheme, this maps to type KSS

Support for HTTPS Proxy Configuration

HTTPS proxy enables accessing public cloud database service as it eliminates the requirement to open an outbound port on a client-side firewall.
Starting with Oracle database 18c, the JDBC drivers support HTTPS Proxy configuration in the connect string, as shown hereafter.

```
(DESCRIPTION=
  (ADDRESS=
       (HTTPS_PROXY=salesproxy)
       (HTTPS_PROXY_PORT=8080)
       (PROTOCOL=TCPS)
       (HOST=sales2-svr)
       (PORT=443))
   (CONNECT_DATA=(SERVICE_NAME=sales.us.example.com)))
```

PERFORMANCE AND SCALABILITY

Getting the best performance and scalability for your Java applications using the Autonomous Databases (ATP and ADW) Cloud services and Oracle databases on-premises includes, as discussed in this blog post: speeding up database connectivity, speeding up SQL statements processing, optimizing network traffic, in-place processing, and scaling out Java workloads.
The most recent performance and scalability enhancements for Java applications in Oracle database release 19c and 18c include: Memoptimized Rowstore, data affinity in RAC environments, the Connection Manager in Traffic Director Mode (CMAN-TDM), and Shard routing APIs

Memoptimized Rowstore

This new Oracle database 19c mechanism allows fast ingest (i.e., high speed ingestion of small amounts of data from a large number of clients simultaneously) and fast lookup (i.e., querying data at a very high frequency).

Assuming a table has been created with the following options

```
CREATE TABLE customers (
            id NUMBER(20,0),
            name VARCHAR2(90 BYTE),
            region VARCHAR2(10 BYTE)
        )
        segment creation immediate
        memoptimize for write
        ;
```

The following SQL INSERT statement via JDBC will perform a fast ingestion of the data i.e., the user call returning very quickly.

```
INSERT /*+ MEMOPTIMIZE_WRITE */ INTO CUSTOMERS VALUES (2, 'JOHN
DOE', 'NORTH');
```

See section 12.5 of the database performance tuning doc for more details on Memoptimized Rowstore.

Oracle Connection Manager in Traffic Director Mode (CMAN-TDM)

CMAN-TDM is an enhancement of the Oracle Connection Manager (CMAN); a database proxy which transparently furnishes the following performance capabilities: statement caching, rows prefetching, result set caching, connection multiplexing. CMAN-TDM furnishes also transparent security and high-availability mechanisms. Starting with the Oracle database release 18c, the JDBC drivers support CMAN-TDM. Java applications get its benefits, transparently.

You can see more on CMAN-TDM on this Web page.

Oracle RAC Data Affinity

In Oracle RAC environment, data in a table may be partitioned (i.e., subset of rows) in such a way that a partition is associated or "*affinitized*" to a specific instance of the RAC system. The localization of "*affinitization*" of data to a RAC instance leads to more scalability.
Starting with the Oracle database 18c, the Oracle Java Connection pool known as Universal Connection Pool (UCP) supports RAC Data affinity as follows:

1) UCP gathers the topology of data partitions
2) Connections requests that need to leverage RAC Data Affinity furnish the affinity key using the database Sharding key and connection builders as

```
PoolDataSource pds = new PoolDataSourceImpl();
 // configure the datasource with the database connection properties.
 OracleShardingKey dataAffinityKey = pds.createShardingKeyBuilder()
 .subkey(1000, OracleType.NUMBER)
 .build();

 Connection connection = pds.createConnectionBuilder()
 .shardingKey(dataAffinityKey)
 .build();
```

See the Oracle Universal Connection Pool doc for more details.

Shard Routing APIs

Java 9 introduces the ShardingKey interface to indicate a Sharding key object; any mid-tier connection pools may route connection requests to specific shards based on the sharding keys.
The Oracle JDBC drivers support the standard Sharding Key API.

However, the ability for each mid-tier to access any arbitrary shard leads to the caching of the entire topology of all sharded databases and the pooling of connections to those shards, by each of these mid-tiers.
To avoid such waste of system resources (i.e., database connections), the Universal Java Connection Pool (UCP) furnishes a routing API that "couples" each mid-tier (or few of them) with a specific Shard. This pattern is known as "swim lanes", where each lane "connects" a web server to a Java application server all the way down to an instance of a Shard.
The benefit of creating an affinity between mid-tiers and Shards, reduces the number of connections each mid-tier has to establish and increases overall system scalability.
Here are the public interfaces and classes

```
/**
 * This class extends the UCP's internal shard routing cache
 *
 */
public class OracleShardRoutingCache extends ShardRoutingCache {
...
/**
   * Creates an instance of a Shard Routing cache to be used
   * by a mid-tier that needs to do shard-based routing.
   * Once this cache is created,getShardInfoForKey can be used for
   * every Sharding Key to know which shard needs to be used.
   */
public OracleShardRoutingCache(Properties dataSourceProps)
      throws UniversalConnectionPoolException {

  }
  ...
  /**
   * Gets the information of each sharded database that maps to
   * the sharding keys.
   */
public Set<ShardInfo> getShardInfoForKey(OracleShardingKey key,
    OracleShardingKey superKey) {

  }
}

/**
 * When the routing cache is queried for shard information, a set of
 * objects of this type is returned.
 * Each object furnishes the required information about one of the shards
 * referred to by the sharding keys.
 */
public interface ShardInfo {

  /**
   * Returns the shard name.
   *
   * @return shard name
   */
  String getName();

  /**
   * Returns the shard's priority.
   *
   * @return shard priority
   */
  int getPriority();

}
```

See this blog post for more details and a working example.

ZERO DOWNTIME

High availability or Zero downtime for Java applications consists in the ability to sustain planned and unplanned outages of the Oracle database. This section concentrates on the latest high-availability

mechanisms in Oracle Database 19c and 18c contributing to zero downtime. These mechanisms include the rolling patching of the embedded JVM (a.k.a. OJVM), Transparent Application Continuity (TAC), Application Continuity (AC) support with the Database Resident Connection Pool (a.k.a. DRCP), and support for legacy JDBC types with TAC/AC. The section does not discuss Oracle database high availability configurations including RAC, RAC One Node and Active Data Guard which are required but not a concern for Java developers. The section describes how the high availability mechanisms come into play to achieve zero downtime.
The previous releases furnished Transaction Guard and Application Continuity that are not described here but used under the covers or have been extended in this release; see the Oracle Database release 19c JDBC doc for more details.

Rolling Patching of the Embedded JVM (OJVM)

Since the release 8i, the Oracle RDBMS furnishes an embedded Java VM (a.k.a. OJVM) for running Java code in the same memory space and process than SQL and PL/SQL. See the OJVM GITHUB examples.
The JVM is made of Java system classes that are shared across all sessions accessing the same database. In clustered database configuration, the Real Application Cluster technology allows several RDBMS instances to simultaneously access the same database including the embedded Java VM system classes.
The challenge for patching the OJVM classes, lies in the fact that all RDBMS instances must be stopped thereby, all Java and non-Java applications.

Since the release 18.c (18.4), OJVM can be patched during a planned maintenance operation in rolling fashion resulting in zero downtime[2] for Java and non-Java applications.

Here are the three steps in a Rolling OJVM patching, in RAC environment (your Cloud or database administrator takes care of this operation, Java developers need only to make sure that the `RETRY_COUNT and RETRY_DELAY` in the connect strings will accommodate the small brownout period.

Step #1: Split the set of RDBMS instances sharing the same database into two groups.

Step #2 On group 1 i.e., the first half, stop and patch the Oracle binary (which has an OJVM part); Java and non-Java activity must be relocated to the other half set. Your DBA knows how to relocate Java service. Java connection pools (at least Oracle UCP and Java containers) know how to stop dispensing connection from instances scheduled for patching (see *Sustaining Planned Database Outage*). Resume these systems after the binary patching however, Java work must not be resumed as the patched Oracle binary is no longer in sync with the JVM system classes (still to be patched). An attempt to use Java from these patched systems will block until step #3 is completed.

Step #3: Bring down the second half and patch the OJVM system classes (iow, replace the old OJVM classes with the new ones). This process takes a few seconds ~ 10 to 15 sec (this is the brownout period for Java across the cluster).
Following the patching the OJVM system classes, Java work can resume on group 1 (blocked sessions will resume) **however, the second group of non-patched Oracle binary cannot use the new OJVM system classes; it must be patched on these systems then these are brought back up.** Java and non-Java work can resume on group #2 as well.

Cloud administrators or DBAs handle such Rolling patching of OJVM however, Java developers and architects need to be aware of the orchestration of the OJVM patching process and provision `retry-count, retry-period` as indicated in the recommended connect string.

Fast Application Notification

[2] There is a few seconds brownout period for Java work, across the cluster.

Traditionally, Java applications are notified of the unavailability of the database service at the expiration of the timeouts associated with the in-flight database operations; however, timeouts are not immediate and unpredictable. Fast Application Notification (FAN) on the other hand furnishes immediate notifications. FAN events and notifications are triggered by the following events: `Node Down, Public Network Down, Instance Down, Instance Up, Service Down, Service Up, Service Member Down, Service Member Up, Database Down, Database Up.`

The `simplefan.jar` furnishes the Java APIs for subscribing and managing FAN events by JDBC drivers, Java connection pools and Java containers. In previous releases, 3rd party connection pools, and containers must explicitly enable and register to receive FAN events. Starting with Oracle JDBC release 18c and 19c, the JDBC driver automatically enables FAN in high-availability database environments (i.e., RAC, ADG) when both `simplefan.jar` and `ons.jar` (which supports the transport system for FAN events) are present in the `classpath`.

Transparent Application Continuity (TAC)

Application Continuity (AC), introduced in Oracle database release 12.1, is an Oracle database high-availability mechanism (capture-and-replay) where the JDBC driver and the RDBMS collaborate to capture in driver memory, all database calls (including SQL statements, parameter assignments, session and transaction states) made in a unit of work a.k.a. "*request*" (typically a Java transaction), so as to replay these in a new session against a good standing database instance upon unplanned outages (network down, host down, RDBMS instance down) or forced termination of sessions during planned maintenance.

Transparent Application Continuity (TAC), introduced in Oracle database release 18c, is an enhancement of Application Continuity (AC). The ultimate goal of Transparent Application Continuity is zero client-side configuration therefore, each new database release pushes the settings from the client-side to the server-side and increases the coverage of session states during capture and replay.

This paper covers TAC only and from Java developers perspective; AC and server-side configuration are not discussed but more details can be found in this Application Continuity white paper.

With AC, the capture of calls is demarcated by the beginning (i.e., checking out a connection from the pool, explicit `conn.beginRequest` call) and the end of the unit of work (i.e., checking in a connection back to the pool, `COMMIT, ROLLBACK,` or explicit `conn.endRequest` call). With TAC, such demarcation is implicitly or transparently enabled by the JDBC driver; a state-tracking infrastructure allows the driver to intersperse "request" boundaries with the running Java application code. The implicit request boundary mechanism can be disabled by setting the `oracle.jdbc.beginRequestAtConnectionCreation` system property to false.

REPLAY DATASOURCE

Java applications looking to use TAC/AC must use the Replay Datasource (`oracle.jdbc.replay.OracleDataSourceImpl`) instead of the vanilla JDBC Data source, to get connections, as illustrated hereafter.
The Replay Datasource of the JDBC driver starts the recording of database calls at the beginning of a unit of work; typically at the beginning of a new transaction i.e., connection checkout (`getConnection()`), or explicit call to `beginRequest()` on an `OracleConnection` object. A Logical Transaction ID (LTXID) is associated to the new transaction; the LTXID is used by Transaction Guard to check whether an in-flight unit of work has committed or not (see the "*Sustaining Unplanned Outage*" section for the orchestration of the various TAC/AC building blocks)

At the end of the unit of work i.e., `COMMIT, ROLLBACK`, connection checked back to the pool (`conn.close()),` or `conn.endRequest`). The Replay datasource stops and drops the recording of

database calls.

```
...
import java.sql.Connection;
import javax.sql.PooledConnection;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.replay.OracleDataSourceFactory;
import oracle.jdbc.replay.OracleDataSource;
import oracle.jdbc.replay.OracleConnectionPoolDataSource;

…
OracleDataSource rds = OracleDataSourceFactory.getOracleDataSource();
Connection conn = rds.getConnection();
((OracleConnection) conn.beginRequest(); // Explicit request begin


…
OracleConnectionPoolDataSource rcpds =
OracleDataSourceFactory.getOracleConnectionPoolDataSource();
PooledConnection pc = rcpds.getPooledConnection(); Connection conn2 =
pc.getConnection(); // Implicit request beginRequest
```

SESSION STATES

TAC requires the database service `FAILOVER_TYPE` attribute to be set to `AUTO`; consequently, the `FAILOVER_RESTORE` attribute is also set to `AUTO`. When `FAILOVER_TYPE` is set to `AUTO`, the JDBC driver automatically starts a new unit of work on connection checkout. The `oracle.jdbc.enableImplicitRequests` property can be set to `FALSE` to turn off the implicit request demarcation (`TRUE` by default).

When `FAILOVER_RESTORE` is set to `AUTO`, TAC restores the following session states during replay: `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_SORT`, `NLS_NUMERIC_CHARACTER`, `NLS_TERRITORY`, `NLS_TIME_FORMAT`, `NLS_TIME_TZ_FORMAT`, `NLS_TIMESTAMP_FORMAT`, `NLS_TIMESTAMP_TZ_FORMAT`, `CURRENT_SCHEMA`, `MODULE`, `ACTION`, `CLIENT_ID`, `ECONTEXT_ID`, `ECONTEXT_SEQ`, `DB_OP`, `AUTOCOMMIT` states, `ERROR_ON_OVERLAP_TIME`, `EDITION`, `SQL_TRANSLATION_PROFILE`, `ROW ARCHIVAL VISIBILITY`, `ROLEs`, and `CLIENT_INFO`.
Note: for Java, `NLS_COMP and CALL_COLLECT_TIME` would not be restored.

The restoration of these states is sufficient for most Java applications however, for restoring custom session states not included in that list, during replay, Java developers may use either the “JDBC Connection Initialization Callback” or the “UCP Connection Labelling Callback” mechanisms; these callbacks and the initial state restoration mechanism are exclusive (i.e., the callbacks override the state restoration mechanism).

```
// Example of JDBC 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
       ...
     }
}
```

Note: the initialization callback is executed every time a connection is borrowed from the pool or during replay; it is idempotent and must not commit or rollback the in-flight transaction.

DATABASE MUTABLES FUNCTIONS

Database mutable functions such as `SYSDATE, SYSTIMESTAMP, SYS_GUID`, and `sequence.NEXTVAL` return new values each time these are called. As a result, during the TAC or AC replay of in-flight unit of work following unplanned outage of the database service, the result sets will differ from the original values and the system will consider that the replay has failed.
If your Java application uses any of these mutable functions, to ensure that during replay, the returned values from these functions remain unchanged (as in the original call), you must request your Cloud or database service administrator to grant the `KEEP` privilege to your database user.

SIDE EFFECTS

A database call from a client-side Java application may incur a call to an external system (RPC) from within the database session such as HTTP callout, FTP callout (file transfer), sending an email. These external call (or side effects) are also replayed by AC. TAC on the other hand, is capable of detecting these side effects and automatically disabling them. If your Java application requires the side effects to be replayed, you must fallback to AC and use the `disableReplay()` call of the `oracle.jdbc.replay.ReplayableConnection` interface.

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

}
```

LEGACY ORACLE JDBC TYPES

The earlier Oracle JDBC releases furnish legacy data types implemented as concrete Java classes that, unlike interface-based implementation `java.sql` types, cannot be proxied during replay (and deprecated per Metalink note 1364193.1).
Starting from Oracle Database Release 18c, JDBC driver supports the following concrete classes with Application Continuity (AC) and Transparent Application Continuity (TAC):
`oracle.sql.CLOB, oracle.sql.NCLOB, oracle.sql.BLOB, oracle.sql.BFILE, oracle.sql.STRUCT, oracle.sql.REF, oracle.sql.ARRAY`

The following concrete classes are not supported: `oracle.sql.OPAQUE, oracle.sql.STRUCT,` and `oracle.sql.ANYDATA`.

Application Continuity (AC) support in DRCP

Starting with the Oracle database 18c release, the Oracle JDBC drivers now support Application Continuity when the Database Resident Connection Pool (DRCP) DRCP is configured on the server-side. The Java connection string determines the use of DRCP as in the following example

```
...
String url = "jdbc:oracle:thin:@(DESCRIPTION =
 (TRANSPORT_CONNECT_TIMEOUT=3000)
 (RETRY_COUNT=20)(RETRY_DELAY=3)(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);
...
```

Zero Downtime Java Checklist

- Use the Replay Datasource (`oracle.jdbc.replay.OracleDataSourceImpl`) of the Oracle JDBC driver (see TAC section above); we are looking into removing such requirement in future releases.
- Get the latest Oracle JDBC drivers and apply the recommended patches to support TAC/AC[3].
- Use the recommended Java connect string with the appropriate values for `CONNECT_TIMEOUT`, `RETRY_COUNT`, `RETRY_DELAY` and `TRANSPORT_CONNECT_TIMEOUT` to avoid that Java applications time out when the database service is temporarily unavailable (for tens of seconds or few minutes).
- Enable JDBC Statement Caching and disable the one in Java containers; this allows the driver to identify statements that are closed at the end of the “request” and free up the memory
- Enable Fast Application Notification (FAN) by ensuring that *`simplefan.jar`* and *`ons.jar`* are in the `classpath`
- Follow the steps for sustaining planned maintenance or unplanned outages, described hereafter.

Sustaining Planned Database Downtime

Planned outages of the database server are sometime required for OS or RDBMS patching, or hardware upgrade. The Cloud or database administrators manage the schedule and the notification for these outages. To shield Java applications from these outages, and ensure continuity, the Java frameworks (driver, connection pools, containers) and the user Java code have to take a couple of steps outlined hereafter.

CLOUD OR DATABASE ADMINISTRATORS STEPS

1. If the the database service is available only on a single system then, relocate it from the system to be maintained to another one. The exact command is not the concern of Java developers.
    - This action sends a “Planned Maintenance FAN event” notification to the JDBC driver

[3] **DB19c JDBC bug fixes for TAC/AC: 29150338, 29195279, 29313347. **DB 18c JDBC bug fixes for TAC./AC: 27748210, 29313347, 28381686, 28504351. **DB19c UCP bug fixes for TAC/AC: 27423500, 29128935. **DB 18c UCP bug fixes for TAC/AC: 27103398, 27290134, 27423500, 27479395, 29325356. **Weblogic bug fixes for TAC/AC: 24919627, 26336757

and/or the Java Connection pool (UCP) or any Java container which has subscribed to receive those events

2. Alternatively, if the database service is also available on other systems then, stop the database in a specific way on the system to be maintained. The exact command is not described here as this is not the concern of Java developers.
    - This action sends a “Planned Maintenance FAN event” notification to the JDBC driver and/or the Java Connection pool (UCP) or Java container which has subscribed to receive those events
3. Allow time for the Java applications to complete in-flight work and relinquish the database connections attached to the system to be maintained. This time is referred to as *drain-period.*
4. Stop the database instance or the system, once all applications have stopped their activities on the system to be maintained.
    - It might be necessary to force terminate the database sessions associated to batch jobs or Java applications that cannot complete the in-flight work or relinquish their connections.
5. Once the maintenance is completed, the service could be relocated back or restarted on the system that has been maintained.

JAVA INFRASTRUCTURE AND JAVA DEVELOPER STEPS

Upon receiving the “Planned Maintenance FAN event” notification, the Java connection pool (UCP) or Java containers (Weblogic, WebSphere, JBoss) take the following action

a) Stop dispensing connections that are attached to the instance of the database service running on the system to be maintained
b) Clean from the pool i.e., remove, idle connections attached to the instance of the database service running on the system to be maintained
c) Let the in-flight work complete, assuming the Cloud service administrator or your DBA have specified a realistic *drain-period*. Java developers too may specify a *drain-period* through the `oracle.ucp.PlannedDrainingPeriod` system property with a non-zero value.

Upon receiving the “Planned Maintenance FAN event” notification, the JDBC driver closes the connections in use when one of the “*Safe-draining APIs*” described hereafter is invoked, so as to allow those connections to be safely removed.

- `java.sql.Connection.isValid(int timeout)`
- `oracle.jdbc.OracleConnection.pingDatabase()`
- `oracle.jdbc.OracleConnection.pingDatabase(int timeout)`
- `oracle.jdbc.OracleConnection.endRequest()`
- `Statement.execute(`any SQL command with the following Hint as the first non-Comment token`)`
  `/*+ CLIENT_CONNECTION_VALIDATION */`
  For example: `/*+ CLIENT_CONNECTION_VALIDATION */ SELECT 1 FROM DUAL`

Java containers including Tomcat, Weblogic, WebSphere, Wildfly, and JBoss have specific connection validation options, such as `TestConnectionOnReserve, PreTest Connection, check-valid-connection-sql, TestonBorrow` -- to be specified in the datasource configuration.

The rationale for these APIs is to indicate to the driver that the Java application has made provision to do something in case the result of the invocation of one of these APIs is False. *If you test the validity of a connection, it means you are prepared to take the appropriate action in the case that the connection is declared Invalid, by the driver or the connection pool, in prevision for the planned maintenanc*e.

User Java code performing in-place processing in the database (e.g., Java stored procedure, other) using the embedded JVM (a.k.a. OJVM) are usually invoked from a top-level call i.e., JDBC `CallableStatement`) or mid-tier tool or utility. These in-database user Java code are covered by the planned maintenance for top-level Java.

In summary, sustaining planned maintenance is generally transparent to Java applications provided that Java developers have configured the latest jars (`ojdbc10.jar or ojdbc8.jar,`

`ucp.jar, simplefan.jar` and `ons.jar`) and implemented the general Java best practices i.e., returning connections back to the pool when not in use as well as the HA recommendations i.e., configure the recommended connect string that supports high availability (see above) and invoke a "Safe-Draining API" for long running transactions or batch jobs.

Long running transactions or batch jobs may be terminated if these have not implemented the best practices. In that case, unplanned outages mechanisms such as the Transparent Application Continuity come to the rescue and replay uncommitted in-flight work on the available members of the service (or when the single instance service resumes).

Sustaining Unplanned Database Outages

THE GENERAL FUNCTIONALITY

Unplanned database outages correspond to one of the following situations: the database system or host is down, the public network connecting the database system is down, the database instance is down (but not the entire system), the database service being used by the Java application is down (but not the entire database instance), and a member of the service is down (but not the entire service).

When any of these situations occurs, the corresponding FAN event is immediately published by one of the surviving instance or service member within your RAC cluster or the disaster recovery site in Active Data Guard configuration. The subscribers i.e., Java drivers, connections pools or containers receive the notification immediately and take the appropriate action, transparently to the user Java code (plain Java, microservice, Servlet, Bean and so on).

As soon as the FAN event is published and received by the subscribers, Transparent Application Continue (TAC) kicks-in, provided that the Java application or framework have configured the latest jars (`ojdbc10.jar or ojdbc8.jar, ucp.jar, simplefan.jar` and `ons.jar`) and the Java application implemented the general Java best practices i.e., returning connections back to the pool when not in use as well as the recommended connect string (see above).

Typically, UCP (or the like) performs the following actions in cooperation with the driver and the database:

1. Hides the SQL exception from the application (assuming this is a recoverable exception; see the JDBC doc for more details).

2. Cleans up the pool by removing orphan connections attached to the failed instance/node/data-center.

3. Creates temporary connections to a good standing instance/node/data-center; then restore the session states automatically preserved by TAC/AC or invokes user-implemented initialization or labelling callback(s). It also re-cast the values of mutable functions, as set in the failed unit of work, assuming the `KEEP` privilege has been granted to the database sessions corresponding to the Java connections in question.

4. Triggers Transaction Guard (TG) under the covers, to check whether an in-flight work has committed (or rolled back), using the logical transaction id (`LTXID`) assigned at the beginning of the transaction; the loss of connectivity with the database could have occurred after the reception of the `COMMIT` or `ROLLBACK` operation by the database session. Transaction Guard guarantees that an un-committed transaction will not/never commit after TG-check thereby making safe to replay.

5. Replays the recordings of in-flight units of work; the recording for each connection that has enabled the `Replay Datasource` is kept in driver memory.
6. Upon replay, if the outcome is identical (i.e., result set checksum) then TAC/AC is successful and the control is given back to the application to continue; if not, the exception is re-cast and visible the user Java code (assuming there is a provision to deal with such exception).

MORE CONSIDERATIONS

TAC/AC works for most Java applications however, there could be situations where TAC/AC will not be able to replay successfully. We recommend to run the ORAchk utility, refer to My Oracle Support note 1268927.2.

CONCLUSION

This paper gives you, Java developer and/or architect, the best practices for getting the best performance, scalability, high-availability, and security from your Oracle database 19c (on-premises and/or in the Cloud.

ORACLE CORPORATION

Worldwide Headquarters
500 Oracle Parkway, Redwood Shores, CA 94065 USA

Worldwide Inquiries
TELE + 1.650.506.7000 + 1.800.ORACLE1
FAX + 1.650.506.7200
oracle.com

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com. Outside North America, find your local office at oracle.com/contact.

 blogs.oracle.com/oracle  facebook.com/oracle  twitter.com/oracle

Integrated Cloud Applications & Platform Services

White Paper **Java Programming with Oracle Database 19c**Java and the Oracle database 19cJava Programming the Oracle database 19cJava Programming The Oracle Database 19c
January 2020
Author: Kuassi Mensah

 Oracle is committed to developing practices and products that help protect the environment