What’s in Oracle Database 12c Release 2 for Java & JavaScript Developers

Oracle Cloud and on-Premise

ORACLE WHITE PAPER | APRIL 2017

Disclaimer

Introduction

Web applications are evolving from monolithic architecture to agile and scalable service or micro-service based architecture. Programming models or styles are evolving from procedural to functional and reactive. Data models and persistence requirements are also evolving from a single model to a polyglot persistence one. With these evolutions come new requirements on database engines in terms of performance, multi-tenancy, high availability, scalability, security, and so on. Building on the strengths of release 12.1[1], the Oracle Database 12c Release 2 comes with a wealth of features addressing these requirements. This paper aims at giving Java and JavaScript[2] developers, a summary of how they can exploit the new capabilities in their applications.

Cloud First

The Oracle Database 12c Release 2 has been released first on the Oracle Cloud[3] catering to various development and deployment requirements. Java applications and IDEs on-premises and on Cloud Services may seamlessly connect to the various Oracle Database cloud services using the Oracle JDBC driver and UCP (the Oracle Java connection pool). See the following pages for more details

http://www.oracle.com/technetwork/database/application-development/jdbc-eecloud-3089380.html
http://www.oracle.com/technetwork/database/application-development/jdbc/index-3093936.html
http://www.oracle.com/technetwork/database/application-development/jdbc/documentation/default-3396167.html

Support for the Latest Java standards

Java SE 8 brings, among other features, lambdas expressions as basis for functional programming, the java.util.streams package, the java.util.concurrent.CompletableFuture for support of asynchronous reactions, enhanced security, and so on. The complete summary of the new Java 8 features can be found @ http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

With Oracle Database 12c Release 2, the JDBC drivers, the Java connection pool (UCP) and the embedded JVM (a.k.a. OJVM), all support Java SE 8 and JDBC 4. 2.

Support for JavaScript

Java SE 8 also brings the Nashorn JavaScript engine as a replacement for Rhino for better performance and features. Java and JavaScript developers can launch Nashorn in the following easy steps.

1. Instantiate a script engine manager

1 http://www.oracle.com/technetwork/database/application-development/12cdb-java-perf-scal-ha-security-1963442.pdf

2 i.e., plain JavaScript; for Node.js please refer to the Node.js Developer Center on the Oracle OTN @ http://www.oracle.com/technetwork/database/database-technologies/scripting-languages/node_js/oracle-node-js-2399407.html

3 https://cloud.oracle.com/database

2. Create a JavaScript engine

3. Pass your resource stream reader as the argument to the eval method of the engine.

And here is the code fragment corresponding to these steps.

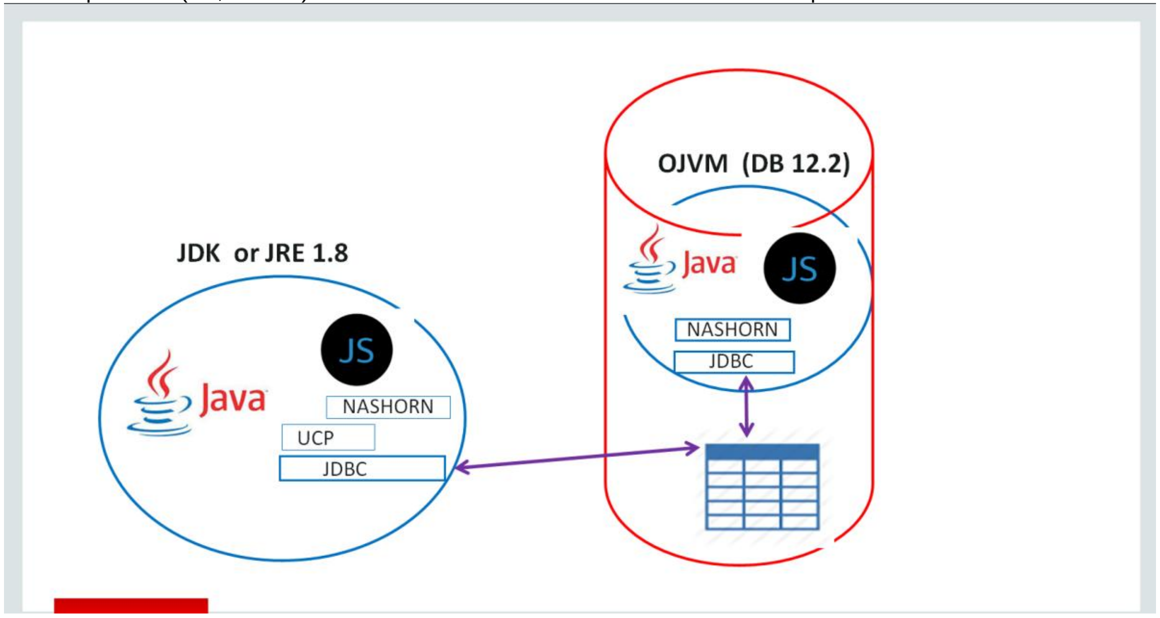
```
import javax.script.*;
import java.net.*;
import java.io.*;
...
// create a script engine manager
ScriptEngineManager factory = new ScriptEngineManager();
// create a JavaScript engine
ScriptEngine engine =
    factory.getEngineByName("javascript");
// create schema resource URL
URL url = Thread.currentThread()

    .getContextClassLoader().getResource("hello.js");

engine.eval(new InputStreamReader(url.openStream()));
```

Database Access from JavaScript

Running JavaScript on the JVM (JDK, JRE and OJVM) allows seamless interoperability between Java and JavaScript; for example, and in the absence of a standard database access API in JavaScript, making JDBC calls from JavaScript. The following picture shows Java and JavaScript using JDBC for table access from Java and JavaScript clients (i.e., outside) the Oracle database as well as Java and JavaScript from within.



JavaScript in the database

The support for the Nashorn engine in OJVM with Oracle database 12.2 allows data-bound JavaScript functions or procedures to run directly in the database, close to data for better performance (i.e., no network latency), reuse of skills, libraries and code. As explained earlier, Nashorn allows JDBC calls within JavaScript, for data access.

The following JavaScript code will run in the embedded JVM in the database (a.k.a. OJVM).

```
var selectQuery = function(id)
{
   var Driver = Packages.oracle.jdbc.OracleDriver;
   var oracleDriver = new Driver();
   var url = "jdbc:default:connection:";
   var query = "";
   var output = "";

   if(id == 'all') {
 query ="SELECT a.data FROM employees a";
   } else {
 query ="SELECT a.data FROM employees a WHERE a.data.EmpId=" + id;
   }
   var connection = oracleDriver.defaultConnection();

   // Prepare statement
 var preparedStatement = connection.prepareStatement(query);

   // execute Query
 var resultSet = preparedStatement.executeQuery();

   // display results
 while(resultSet.next()) {
  output = output + resultSet.getString(1) + "
";
 }

   // cleanup
    resultSet.close();
    preparedStatement.close();
    connection.close();
    return output;
}
```

How exactly would you run JavaScript in your Oracle database?

1. First, load your JavaScript code in your database schema as a Java resource, using the loadjava utility.
2. Then use one of the three invocations possibilities:
   (i) SQL> `call dbms_javascript.run() from SQL or PL?SQL;`
   (ii) `call DbmsJavaScrpt.Run()` from Java in the database
   (iii) use the `javax.script` API.

The following blog post @ http://db360.blogspot.com/2016/11/javascript-in-oracle-database-12c.html describes each alternative in details,

We will post the JavaScript/Nashorn code samples on GitHugb, under the oracle master repository `https://github.com/oracle/oracle-db-examples/tree/master/javascript`

Performance and Scalability

This release brings new performance and scalability enhancements to Java and JavaScript applications through JDBC, UCP and OJVM.

OJVM

The rationale for running PL/SQL and JVM languages including Java, JRuby, Jython, Closure, Scala, and now JavaScript, in the database is performance. In this release, the embedded JVM (a.k.a. OJVM) has been enhanced with a more efficient JIT and memory management for better performance. My good friend Marcelo Ochoa has tested the DB 12.2 OJVM and posted the following blog @ http://marceloochoa.blogspot.com.ar/2016/11/12cr2-versus-12cr1-scan-time-using-java.html shows OJVM performance improvement in release 12.2,

JDBC and UCP

The Oracle JDBC driver now supports compressing data on the wire resulting in improved response for client/server communication over WAN. The feature is enabled by setting the oracle.net.networkCompression property as illustrated in the following code fragment.

```
// Enabling Network Compression
   prop.setProperty("oracle.net.networkCompression","on");
// Optional configuration for setting the client compression threshold.
   prop.setProperty("oracle.net.networkCompressionThreshold","1024");
   ds.setConnectionProperties(prop);
   ds.setURL(url);
   Connection conn = ds.getConnection();
```

The Oracle Universal Connection Pool (UCP) has been redesigned using wait-free and multidimensional K-D Trees search[4], resulting in the ability to sustain over 300.000 pool operations per second without degradation.
UCP in DB 12.2 also allows configuring the frequency of the connection health check thereby reducing the overhead of systematic health check at connection check-out.
Here is the Java API:

```
public void setSecondsToTrustIdleConnection(int secondsToTrustIdleConnection)
                         throws java.sql.SQLException
```

If secondsToTrustIdleConnection is set to 30, then the connection is assumed valid for the next 30 seconds.

Java Shared Pool for Sharded Databases

Oracle Database 12c Release 2 introduces the Sharded Database architecture; it enables horizontal data partitioning across hundreds of databases, resulting in unlimited scalability. A Shard Director maintains a map of shards and sharding keys. Both Oracle JDBC and UCP support the Sharded Database architecture through new APIs for building the shard keys and super-sharding keys, as illustrated in the following Java code fragment.

4 See the related JavaOne session https://www.youtube.com/watch?v=PEwBrjkJfrk&t=352s&list=PLPIzp-E1msrYicmovyeuOABO4HxVPlhEA&index=59

```
// A sharding key make of 2 sub-keys
   OracleShardingKey shardingKey =
                    dataSource.createShardingKeyBuilder()
                   .subkey("Customer_EMAIL",oracle.jdbc.OracleType.VARCHAR2)
                   .subkey("1234", oracle.jdbc.OracleTypes.NUMBER)
                   .build();
```

Shard-aware Java applications may now request connections to a specific shard.

```
  // Request a connection to a shard
     Connection conn =
                    pds.createConnectionBuilder()
                       .shardingKey(shardingKey)
                       .superShardingKey (superShardingKey)
                       .build();
```

UCP pulls the Sharding topology from the Oracle Database Shard Director and routes the connection requests to the appropriate shard, directly, thereby avoiding the extra hop (i.e., going to the Shard Director).

Java Shared Pool for Multitenant Databases

In this release, UCP furnishes a shared pool and repurposes free connections attached to one pluggable database (a.k.a. PDB) for use by another one. A single connection pool can now be shared across a large number of tenants, each with its own PDB. Thousands of tenants can now each have their own datasource or database service and scale at large. Here is a Java code fragment where each tenant is connecting through it’s own database service.

```
PoolDataSource multiTenantDS = PoolDataSourceFactory.getPoolDataSource();
multiTenantDS.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
// Set the common user for the CDB and point to the root service
multiTenantDS.setUser("c##1");
multiTenantDS.setPassword("password");
multiTenantDS.setURL("jdbc:oracle:thin…");
// Create a connection to Tenant-1
Connection pdb1_conn = multiTenantDS.createConnectionBuilder()
                        .serviceName("pdb1")
                        .build();

// Get the password enabled role for tenant-2, if configured on the database
Properties pdb2Roles = new Properties();
pdb2Roles.put("pdb2-role", "pdb2-password");

// Create a connection to Tenant-2 and apply the tenant specific PDB roles.
Connection pdb2_conn = multiTenantDS.createConnectionBuilder()
                         .serviceName("pdb2")
                         .pdbRoles(pdb2Roles)
                         .build();
```

PL/SQL Callback Interface

Applications need sometimes to assign new values to session states such as NLS_CURRENCY and CURRENT_SCHEMA, at runtime. A new PL/SQL Callback mechanism allows setting new values to those states without incurring roundtrips to the database. Java applications must implement such callback in the RDBMS server then register the Callback through JDBC call. See the Oracle JDBC guide for an example of implementation.

Oracle Update Batching Desupported

The Oracle update batching has been deprecated in Oracle database release 12.1 and de-supported in this release. Consequently, setting the batch size has been disabled and defaults to 1. Applications that are still using the Oracle Update batching API will experience a performance drop (i.e., one row at a time). Oracle recommends using the standard JDBC batching API, instead.

High Availability - Continuous Operations

JDBC -UCP

High-availability and continuous operations are crucial for Web applications and Cloud services. In this release, JDBC and UCP extend their support for high-availability; the new enhancements include:

- Extending the Java APIs for handling FAN events UP, DOWN and Load Balancing advisories in `SimpleFan.jar`; with these APIs, third party Java containers or frameworks can subscribe to key FAN HA events and manage service/node failure or addition.
- Support for planned maintenance (i.e., DOWN event, reason=User) in the Oracle JDBC driver; upon receiving the FAN DOWN event, the driver closes all connections attached to the node or service scheduled for maintenance, at a "*Safe Place*". The Java applications must invoke either `isValid()or, isUsable()or, isClosed()` methods on the connection object; alternatively, the application may issue the following SQL call `SELECT 1 FROM DUAL /*+ CLIENT_CONNECTION_VALIDATION */`.
  See the JDBC documentation for more details.
- Extending Transaction Guard (TG) to XA datasources. Transaction Guard for Java is an HA API introduced in Oracle Database 12c Release 1 for a reliable determination of the outcome of the COMMIT statement upon a break in communication between the application and the RDBMS. In this release, Transaction Guard has been extended to XA transactions with one-phase commit optimization, read-only optimization, and promotable XA.
- Extending Application Continuity (AC) to XA datasources. Application Continuity for Java is an HA solution introduced in Oracle Database 12c Release 1 to ensure service continuity in the face of unplanned database, service or node outages. It requires the use of the JDBC Replay Datasource and the proper configuration of the database service. In this release, AC supports XA datasources as well.
- By pulling the Shard topology and acting as a Shard Director, UCP allows Shard-aware Java applications to work and request connections to specific Shards, even if/when the Shard Director is down.

Best Practices

The high-availability best practices include:

a) **For planned maintenance**: Use Oracle Database HA configurations (RAC, ADG, GDS, Multi-tenant); use Oracle driver or connection pools for seamless HA support; use service for location transparency; properly configure the connection strings with the latest capabilities; ensure that FAN events are auto-configured and auto-enabled (a 12.2 feature) to trigger draining by the driver or the pool; properly set service attributes including `drain_timeout(in seconds)` and `stopoption(immediate, transactional).`

b) **For unplanned outages**: follow the best practices for planned maintenance, above. In addition: configure Application Continuity; use Oracle connection pools (i.e., UCP) or add request boundaries to database units of work (a.k.a. Requests); return the connections back to the pool when not in use; run `ORAChk` to identify whether JDBC concrete classes are used by the application or not; grant keeping mutable values privileges, e.g. seq.nextval, sysdate, systimestamp to the user schemas used by the applications.

Security

The following security enhancements are new in this release.

The JDBC driver in DB 12.2 supports SSL v1.2 (a.k.a. TLS v1.2) by default.
For encryption and enhanced security, Java connectivity to the Oracle Database Exadata Express Cloud Service requires Java Key Store (JKS) files.
For running JavaScript in the database using Nashorn in OJVM, the database schema must be granted the DBJAVASCRIPT role. In addition, the use of `javax.script` API (JSR223) is restricted to sandboxed mode only, with a minimal set of permissions.
For Web Services Callout (see later), the database schema must be granted the OJVMWCU role.
The new Web Services Call-Out Utility (see later) provides support for SSL based Web services, in addition to basic HTTP authentication.

Manageability, Ease of use, Diagnosability

New in this release: OJVM support for Web Services Callout (REST and SOAP), runtime logging at feature level (JDBC), OJVM support for long identifiers, JDBC support for PL/SQL Boolean, JDBC support for feature level debugging, enhancements to debugging Java code running in OJVM, and enhancements to the Database Resident Connection Pool (DRCP). Some of these enhancements are described briefly in the following sections.

OJVM - Web Services Callout

The ability to invoke remote/external SOAP Web Services and retrieve data was a popular feature, enabled by JPublisher . Following its de-support in DB 12.1, a new utility is now available in DB 12.2; in addition, we've added support for RESTful Web Services call out.
The new Web Services Callout utility accepts WSDL (SOAP Web Services) as well as WADL (Restful Web Services) and additional parameters then retrieves the Web Services client proxy, loads it in the database and generate the wrapper for SQL and PL/SQL calls.
See the doc http://docs.oracle.com/database/122/JJDEV/database-as-web-service-consumer.htm#JJDEV13475 for more details.

OJVM - Long Identifiers and Debugger

OJVM now supports long identifiers i.e., the maximum length of a SQL identifier is now 128 bytes.

Debugging Java in the database through JDWP interface allows attaching a debug session to a connection, setting or clearing breakpoints, stepping through the Java code, setting and changing the values of variables, evaluating expressions, and setting or clearing watch-points.

JDBC – UCP

This release also brings new enhancements to the Database Resident Connection Pool (DRCP) including: the Multi-Property Labelling, new statistics views and AWR reports for performance monitoring and tuning, a new MAX_TXN_THINK_TIME property for pooled servers with transactions in progress, a PL/SQL callback for session state fix-up, and the ability to share proxy sessions.

JDBC now supports feature level debugging through the `OracleDiagnosabilityMBean`; this consists in enabling and disabling logging for selected features, at runtime. For example, logging can be enabled for Fast Connection Failover feature and disabled for Load Balancing feature. By default, runtime logging is enabled for all features.
See more details @ http://docs.oracle.com/database/122/JJDBC/JDBC-diagnosability.htm#JJDBC-GUID-193E4D19-40D4-40D8-89C5-1A792E15F538

JDBC support for PL/SQL Boolean as parameter bind variable is illustrated by the following code fragment

```
 cstmt.registerOutParameter(1, OracleTypes.PLSQL_BOOLEAN);
   // Execute the callable statement
    cstmt.execute();
    boolean  TF = cstmt.getBoolean(1);
```

Conclusion

This paper discussed the key benefits for Java and JavaScript developers in Oracle Database 12c Release 2.
It walked you through Java connectivity for database in the cloud, the support for the latest Java standards in JDBC, UCP and OJVM, JavaScript with Nashorn database access using JDBC, JavaScript execution in the database with Nashorn in OJVM, the performance and scalability enhancements in JDBC, UCP and OJVM, finally, the security, manageability and ease of use enhancements.
With Oracle database 12c Release 2 in the Cloud and on-premises, Java and JavaScript developers can design and deploy modern Web applications with mid-tier and database-embedded components using functional and reactive programming models while ensuring performance, scalability, high-availability, security, manageability and diagnose-ability.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

CONNECT WITH US

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

Integrated Cloud Applications & Platform Services