Business / Technical Brief

What's in Oracle Database 21c for Java Developers?

New enhancements to JDBC, UCP and OJVM for Designing and Deploying Mission Critical and Cloud Native Java Applications.

Updated: August 2021

Table of contents

Note: A Table of Contents (TOC) is recommended for documents that are more than 10 pages in length. If your statement of direction is shorter, you may remove the TOC page. To remove both the TOC and the page that it appears on, first display hidden characters by clicking on the Paragraph symbol on the Home toolbar. It is a small, square icon that appears to the left of the Quick Style Gallery. Notice the page break displayed as a line at the bottom of this page. Next, highlight all the text on this page and press the Delete key once to remove all the text. Then, highlight the page break and press the Delete key until the page disappears and your cursor is on the first page of body text. The TOC will update semi-automatically with the Heading 1 and Heading 2 styles when you 1) right click on the TOC and then 2) click Update Field in the contextual menu. You may also manually edit the TOC by placing your cursor within the text. Delete this note before publishing.

Introduction

As a Java developer or architect, why would you consider the Oracle database 21c release?
The new enhancements in the RDBMS, the embedded JVM (a.k.a. OJVM), the JDBC drivers, the Java connection pool, and the Oracle Cloud Infrastructure aim at: simplifying the onboarding of those new to Oracle database; improving the experience of those familiar to it; and simplifying the development and deployment of mission critical and Cloud native Java applications.

This technical brief discusses how the new capabilities fulfill the goals outlined above; specifically:
1. The enhancements to Cloud database connectivity
2. The support for popular Java frameworks and IDEs for simplifying Java developers experience or onboarding with the Oracle database
3. The support for Cloud native applications (Data-driven MicroServices and Serverless functions)
4. The enhancements to diagnosability and tracing for improving developers experience and mission critical deployments
5. The performance and scalability enhancements for mission critical deployments including: the async/reactive extensions, support for virtual threads (project Loom), the Reactive Streams Ingestion library, support for GraalVM Native Image, and a new Sharding datasource
6. The Zero Downtime enhancements for mission critical deployments including: Transparent Application Continuity, Zero Brownout OJVM Rolling patching

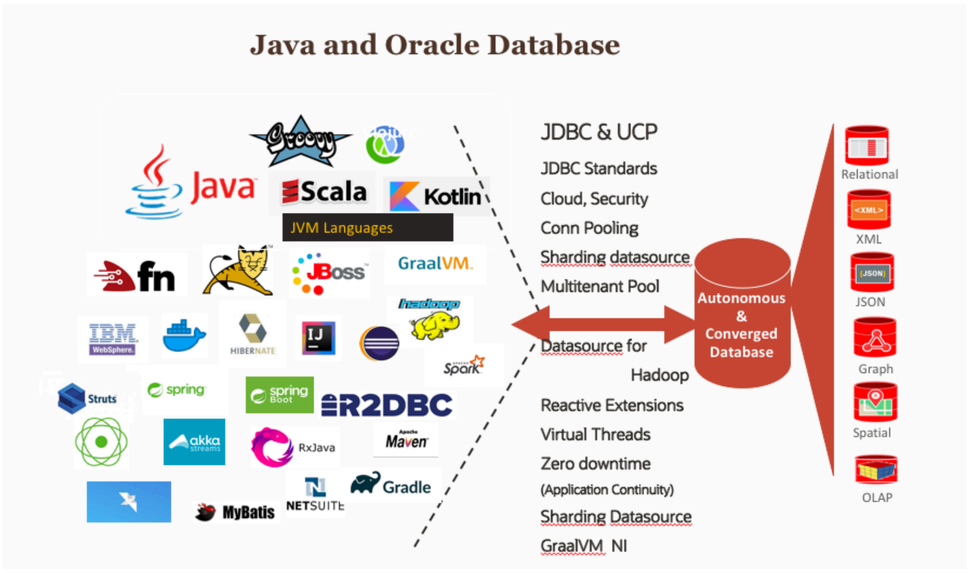


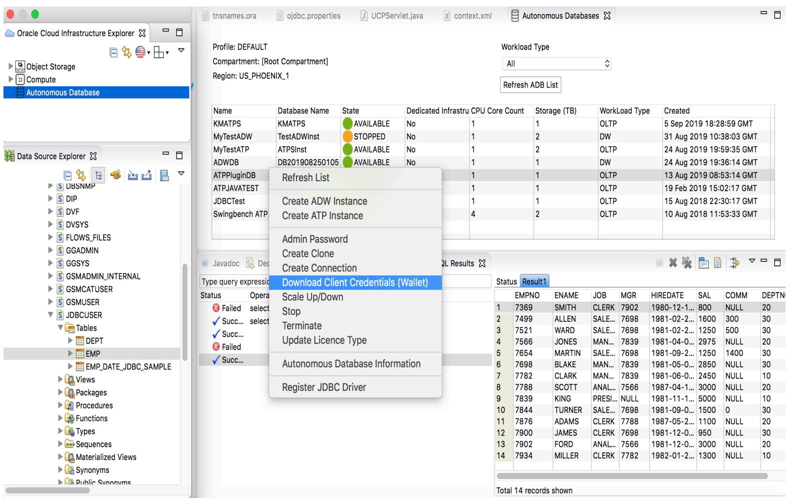
Figure 1 Global view of Java and the Oracle database

Support for Popular Java Frameworks and IDEs

There are tons of Java framework; we looked at the most popular and how to configure these to work efficiently with the Oracle JDBC driver and connection pool (UCP).

Eclipse and IntelliJ* Plugins

The Eclipse plugin is available @ https://github.com/oracle/oci-toolkit-eclipse. See the related blog post for more details.
* The IntelliJ plugin is in the process of being released, as of this write up.



Profile: DEFAULT
Compartment: [Root Compartment]
Region: US_PHOENIX_1

Workload Type: All

Name	Database Name	State	Dedicated Infrastru	CPU Core Count	Storage (TB)	WorkLoad Type	Created
KMATPS	KMATPS	AVAILABLE	No	1	1	OLTP	5 Sep 2019 18:28:59 GMT
MyTestADW	TestADWInst	STOPPED	No	1	2	DW	31 Aug 2019 10:38:03 GMT
MyTestATP	ATPSInst	AVAILABLE	No	1	2	OLTP	24 Aug 2019 19:59:35 GMT
ADWDB	DB201908250105	AVAILABLE	No	1	1	DW	24 Aug 2019 19:36:14 GMT
ATPPluginDB				1	1	OLTP	13 Aug 2019 08:53:14 GMT
ATPJAVATEST				1	1	OLTP	19 Feb 2019 15:02:17 GMT
JDBCTest				1	1	OLTP	15 Aug 2018 22:30:17 GMT
Swingbench ATP				4	2	OLTP	10 Aug 2018 11:53:33 GMT

Type query expressio

Status	Opera
Failed	select
Succ...	select
Succ...	
Failed	
Succ...	

Status Result1

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTN
1	7369	SMITH	CLERK	7902	1980-12-1...	800	NULL	20
2	7499	ALLEN	SALE...	7698	1981-02-2...	1600	300	30
3	7521	WARD	SALE...	7698	1981-02-2...	1250	500	30
4	7566	JONES	MAN...	7839	1981-04-0...	2975	NULL	20
5	7654	MARTIN	SALE...	7698	1981-09-2...	1250	1400	30
6	7698	BLAKE	MAN...	7839	1981-05-0...	2850	NULL	30
7	7782	CLARK	MAN...	7839	1981-06-0...	2450	NULL	10
8	7788	SCOTT	ANAL...	7566	1987-04-1...	3000	NULL	20
9	7839	KING	PRESI...	NULL	1981-11-1...	5000	NULL	10
10	7844	TURNER	SALE...	7698	1981-09-0...	1500	0	30
11	7876	ADAMS	CLERK	7788	1987-05-2...	1100	NULL	20
12	7900	JAMES	CLERK	7698	1981-12-0...	950	NULL	30
13	7902	FORD	ANAL...	7566	1981-12-0...	3000	NULL	20
14	7934	MILLER	CLERK	7782	1982-01-2...	1300	NULL	10

Total 14 records shown

Figure 2 Oracle Cloud Infrastructure Toolkit for Eclipse

The features set for both plugins includes: provisioning new autonomous databases, start/stop/clone, scale up/down OCPU and Storage, change the ADMIN password; downloading the Cloud credentials; testing the connection to the Cloud database; browsing the database schema; and performing other database operations.

Support for Popular Frameworks and Java EE App Servers

1. Maven and Gradle are the more popular build automation frameworks used by Java developers. We now have Oracle JDBC on Maven Central @ https://bit.ly/33bpLVJ; the developers guide @ https://bit.ly/2IBDXjJ gives more details. Starting with the 19.8 release, we've also added the pre-established dependencies (a.k.a. "flavor POMs") to our BOM file so as to allow Spring projects (i.e., Spring Initializr) to pull these.
2. Watch this video for Apache Tomcat Servlet engine connectivity to the Autonomous database ATP.
3. Apache Hadoop: the Oracle Datasource for Hadoop turns Oracle database tables into Hadoop external tables.
4. Hibernate: see how to configure Hibernate and with UCP @ https://bit.ly/32WrYUN
5. MyBatis: see how to configure MyBatis and with @ https://bit.ly/2IRgpGY
6. R2DBC: an open source Oracle-R2DBC driver v0.3.0 is available on Github at https://github.com/oracle/oracle-r2dbc/releases/tag/0.3.0 as well as on central maven at this location.
7. GraalVM: the JDBC driver has been instrumented for GraalVM native image; see more details later in this write-up.

UCP as a Spring DataSource

The Java connection pool (UCP) can now be configured as a Spring datasource with no extra code. Spring retrieves the configuration from the application's properties file (`application.properties`) and autowires (injects) the values to the datasource.

```
spring.datasource.url=jdbc:oracle:thin:@host:1521/myservice
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
spring.datasource.type=oracle.ucp.jdbc.UCPDataSource
spring.datasource.ucp.connection-factory-class-
name=oracle.jdbc.replay.OracleDataSourceImpl
spring.datasource.ucp.sql-for-validate-connection=select * from dual
spring.datasource.ucp.connection-pool-name=connectionPoolName1
spring.datasource.ucp.initial-pool-size=15
spring.datasource.ucp.min-pool-size=10
spring.datasource.ucp.max-pool-size=30
```

See the sample code on how to configure UCP with SpringBoot @ https://bit.ly/SpringBootApp

UCP as JBOSS DataSource

In this release, the Java Universal Connection Pool (UCP) furnishes a class for its integration with JBoss for use by Java EE components (Servlets, JSP, JMS, EJB and so on).

- The class implements the `ServletContextListener` interface and supports method override `@Override public void contextInitialized(ServletContextEvent contextEvent).`
  ```
  @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent)
  ```
- The class is loaded by specifying in `web.xml` using the `<listener>` and `<listener-class>` tags as shown below.

  ```
  <!--Register the UCPServletContextListener explicitly -->
  <listener>
  <listener-class>oracle.ucp.jdbc.UCPServletContextListener</listener-class>
  </listener>
  ```

- The class is invoked when the web.xml is loaded. The object reads a configuration/description file, creates a datasource with those values and binds it to a JNDI address or to an application object injected with CDI. The JNDI address and application object can be used by components to retrieve the datasource and connections. The parameters are retrieved either from the web descriptor (`web.xml`) as `<context-param>` values or using UCP XML configuration file.

  Refer to `oracle.ucp.jdbc.UCPServletContextListener` class for the supported list of UCP properties.

Here is a working sample of the `ServletContextListener` implementation, a snippet of the web descriptor and its use from a Servlet.

Using Web descriptor (web.xml)

```
<!--JNDI datasource name used in application servlet for lookup-->
      <context-param>
            <param-name>ucp.jndiName</param-name>
            <param-value>java:/datasources/mypool_usingwl</param-value>
      </context-param>

<!-- Set UCP connection pool properties here, Refer to javadoc of
UCPServletContextListener to see the list of supported UCP pool properties -->
      <context-param>
            <param-name>ucp.URL</param-name>
            <param-value>jdbc:oracle:thin:@myhost:5521/myservice</param-value>
      </context-param>
      <context-param>
            <param-name>ucp.connectionFactoryClassName</param-name>
            <param-value>oracle.jdbc.replay.OracleDataSourceImpl</param-value>
      </context-param>
      <context-param>
            <param-name>ucp.dataSourceName</param-name>
            <param-value>myDataSource</param-value>
      </context-param>
      <context-param>
            <param-name>ucp.user</param-name>
            <param-value>scott</param-value>
      </context-param>
      <context-param>
            <param-name>ucp.password</param-name>
            <param-value>*****</param-value>
      </context-param>
        . . .
</web-app>
```

Using UCP XML Configuration

To use the UCP XML Configuration file, set the `oracle.ucp.jdbc.xmlConfigFile` JVM system property as follows:

```
      -Doracle.ucp.jdbc.xmlConfigFile=file:/Users/scott/conf/ucp_config.xml
```

Then set the `ucp.dataSourceNameFromXMLConfig` parameter in the web descriptor (web.xml); it must match the data-source-attribute in /ucp-properties/connection-pool/data-source/data-source-name.

Web.xml

```
      <!-- Specify the datasource name as used in ucp_config.xml -->
      <context-param>
         <param-name>ucp.dataSourceNameFromXMLConfig</param-name>
         <param-value>myDataSource</param-value>
      </context-param>

      <!--JNDI datasource name used in application servlet for lookup-->
      <context-param>
          <param-name>ucp.jndiName</param-name>
          <param-value>java:/datasources/mypool_usingwl</param-value>
      </context-param>
```

ucp_config.xml

```
<ucp-properties>
    <connection-pool
    connection-factory-class-name="oracle.jdbc.replay.OracleDataSourceImpl"
    connection-pool-name="pool1"
    initial-pool-size="10"
    max-connections-per-service="15"
    max-pool-size="30"
    min-pool-size="2"
    password="*****"
    url="jdbc:oracle:thin:@myhost:5521/myservice”
    user="scott">
    <connection-property name="autoCommit" value="false"></connection-property>
    <connection-property name="oracle.net.OUTBOUND_CONNECT_TIMEOUT" value="2000">
    </connection-property>
    <data-source data-source-name="myDataSource" description="pdb1" service="ac">
    </data-source>
    </connection-pool>
</ucp-properties>
```

```
Sample Servlet Retrieving the datasource through CDI.
      @Inject
      @UCPResource
      private DataSource ds;

Sample Servlet Retrieving the datasource through JNDI.
      @WebServlet("/OracleUcp")
      public class OracleUcp extends HttpServlet {
        private DataSource ds = null;

        // Retrieve Datasource reference using JNDI
        @Override
        public void init() throws ServletException {
          Context initContext;
          try {
            initContext = new InitialContext();
            ds = (DataSource)
      initContext.lookup("java:/datasources/mypool_usingwl");

Here is a sample Servlet using the datasource.
      protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

          // Retrieve connection from the pool
          try (Connection conn = ds.getConnection();
              Statement st = conn.createStatement()) {
            ResultSet rs = null;
            rs = st.executeQuery("select empno, ename, job from emp");
```

Known limitation: the management and monitoring tools provided from the container cannot be applied to the pool

Enhancements to support Cloud Native Applications

JDBC and UCP Configuration with MicroServices Frameworks

- **SpringBoot**: See how to configure the Oracle JDBC drivers and UCP in a SpringBoot project @ https://bit.ly/SpringBootApp
- **Helidon**: both Helidon MP and SE provide support for Oracle UCP datasources thus inheriting all of the features it provides. Helidon provides various integration and convenience features from automatically injecting UCP datasources into microservices to simplified use in Kubernetes and ATP environments to use of Oracle DB features (including those directly related to micro services architectures).
  See how to configure the Oracle JDBC drivers in a Helidon project @ https://bit.ly/2IP18a5
- **Micronaut**: See how to configure the Oracle JDBC drivers in a Micronaut project @ https://bit.ly/2KEw4KX
- **Quarkus**: See how to configure the Oracle JDBC drivers in a Quarkus project @ https://bit.ly/2J6dYR9

Loading Wallet From Non-File bases System

In this release, a new API for setting an SSLContext on an OracleDataSource gives developers full control over how to load wallets i.e., from memory or non-file based system.
The following code snippets illustrates how to use such API.

```
static SSLContext createSSLContext()
    throws GeneralSecurityException, IOException {
    TrustManagerFactory trustManagerFactory =
      TrustManagerFactory.getInstance("PKIX");
    KeyManagerFactory keyManagerFactory =
      KeyManagerFactory.getInstance("PKIX");

    trustManagerFactory.init(loadKeyStore());
    keyManagerFactory.init(loadKeyStore(), null);

    SSLContext sslContext = SSLContext.getInstance("SSL");
    sslContext.init(
      keyManagerFactory.getKeyManagers(),
      trustManagerFactory.getTrustManagers(),
      null);
    return sslContext;
  }

  static KeyStore loadKeyStore()
    throws IOException, GeneralSecurityException {

    // This example is using a standard file-based source however, I can be reused
    // to load the keystore using an InputStream from a non-file based source.
    try (InputStream keyStoreStream =
          Files.newInputStream(Paths.get("cwallet.sso"))) {
      KeyStore keyStore = KeyStore.getInstance("SSO", new OraclePKIProvider());
      keyStore.load(keyStoreStream, null);
      // keyStore.load(KeyStore.LoadStoreParameter) could be used here as well.
      return keyStore;
    }
  }
```

Security Enhancements in the Previous Release

The following security enhancements were made in the previous release.

- Support for HTTPS Proxy Configuration
- Automatic Provider Resolution (OraclePKIProvider)
- Setting server's domain name (*oracle.net.ssl_server_cert_dn*) in the URL
- Support of new wallet property (my_wallet_directory) in the URL
- Support for Key Store Service (KSS)

JDBC Support for Native JSON Datatype

The Oracle database 21c release furnishes a native JSON datatype and an Autonomous JSON Cloud service . The `oracle.sql.json` package furnishes Java APIs for: accessing the native JSON type values and their storage in binary format; creating, querying and modifying JSON type values; encoding or decoding JSON type values in the same binary JSON format used by the database; converting JSON type values to and from JSON text; binding and accessing JSON type values using the JSON-P interfaces e.g., `javax.json.*.`

Here is a JDBC code sample illustrating the use of JSON Native datatype.

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
import oracle.sql.json.OracleJsonFactory;
import oracle.sql.json.OracleJsonObject;

public class JsonExample {

  public static void main(String[] args) throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    ds.setURL("jdbc:oracle:thin:@myhost:1521:orcl");
    ds.setUser(<user>);
    ds.setPassword(<password>);
    OracleConnection con = (OracleConnection) ds.getConnection();

    // create a table with a JSON column and insert one value
    Statement stmt = con.createStatement();
    stmt.executeUpdate("CREATE TABLE fruit (data JSON)");
    stmt.executeUpdate("INSERT INTO fruit VALUES ('{"name":"pear","count":10}')");

    // create another JSON object
    OracleJsonFactory factory = new OracleJsonFactory();
    OracleJsonObject orange = factory.createObject();
    orange.put("name", "orange");
    orange.put("count", 12);

    // insert the orange object
    PreparedStatement pstmt = con.prepareStatement("INSERT INTO fruit VALUES (:1)");
    pstmt.setObject(1, orange, OracleType.JSON);
    pstmt.executeUpdate();
    pstmt.close();

    // retrieve the pear object
    ResultSet rs = stmt.executeQuery("SELECT data FROM fruit f WHERE f.data.name = 'pear
'");
    rs.next();
    OracleJsonObject pear = rs.getObject(1, OracleJsonObject.class);
    int count = pear.getInt("count");

    // create a modifiable copy of the pear object
    pear = factory.createObject(pear);
    pear.put("count", count + 1);
    pear.put("color", "green");

    // update the pear object
    pstmt = con.prepareStatement("UPDATE fruit f SET data = :1 WHERE f.data.name = 'pear
');
    pstmt.setObject(1, pear, OracleType.JSON);
    pstmt.executeUpdate();
    pstmt.close();

    rs.close();
    stmt.close();
    con.close();
  }
}
```

Diagnosability and Tracing Enhancements

The Connection Identifier is a new diagnosability enhancements in 21c release.

Connection Identifier for Improved Diagnosability

Oracle JDBC Thin driver generates a unique CONNECTION_ID for each connection to the Oracle Database Server. If CONNECTION_ID_PREFIX is configured then it gets appended internally to the system generated CONNECTION_ID value. The CONNECTION_ID_PREFIX must be an 8-byte alphanumeric identifier limited to the following [a...z] [A...Z] [0...9] _ character set. Configuring CONNECTION_ID_PREFIX is optional.

The RDBMS allows assigning a connection id to each connection or group of connections.

1. It can be configured as part of connection string as a name/value pair in the JDBC connect string (`CONNECTION_ID_PREFIX =<value>`) or it can be also configured via connection properties using *"oracle.net.connectionIdPrefix"*.

Example :
(DESCRIPTION= (ADDRESS_LIST= (ADDRESS=...) (ADDRESS=...)) (CONNECT_DATA= (SERVICE_NAME=sales.us.example.com) ((CONNECTION_ID_PREFIX=SALES_PR)))

2. The connection identifier (CONNECTION_ID_PREFIX) will appear in logs

```
 “Exception in thread "main" java.sql.SQLRecoverableException: ORA-
12506, TNS:listener rejected connection based on service ACL filtering
(CONNECTION_ID=SALES_PRovDT/bCGfJngU602xAph8g==)”
```

DMS Metrics and ClientInfo

When present in the `classpath`, the Dynamic Monitoring System (DMS) jars (`ojdbc8dms.jar`, `ojdbc11dms.jar`) furnish diagnosability metrics and limited support for `java.util.logging`.

JDBC furnishes `setClientInfo()` and `getClientInfo()` methods for tagging applications with the end-to-end metrics including: `Actions, ClientId, ExecutionContextId, Module and State`.

Performance and Scalability for Mission Critical Deployment

The following new capabilities will significantly improve the performance and scalability of mission critical Java applications: the JDBC reactive extensions, the support for virtual threads in the driver, the reactive streams ingestion library, driver configuration for GraalVM native image, and the Sharding datasource.

Driver Support for Virtual Threads

Project Loom aims at “reducing the complexity of creating and maintaining the high-throughput concurrent applications” using inexpensive “virtual threads”, delimited continuations and tail-call elimination. The promise of virtual threads is that synchronous calls can perform as well as asynchronous calls.
DB21c JDBC has been instrumented (native method calls, intrinsic locking with synchronized) to support synchronous database access with inexpensive threads i.e., standard JDBC calls using virtual threads. In other words, no need to refactor simple blocking JDBC calls into complex reactive streams calls.
Here is a code snippet illustrating the use of Virtual Threads with JDBC

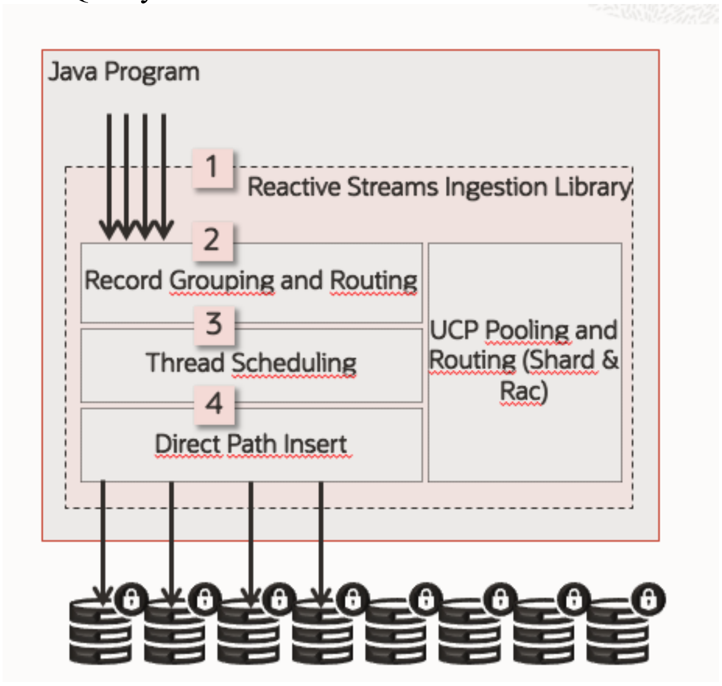
```
// Set the task executor to use Virtual Threads, or to use kernel threads.
final Executor taskExecutor =
useVirtualThreads
  ? newVirtualThreadExecutor(kernelThreadExecutor)
  : kernelThreadExecutor;
…

// Execute a number of JDBC calls with the executor
for (int i = 0; i < taskCount; i++) {
taskExecutor.execute(() -> {
  executeSql("SELECT * FROM emp");
  completionLatch.countDown();
});
}
…
```

```
// Periodically prints a message until all tasks are complete
awaitCompletion(useVirtualThreads, taskCount, completionLatch);
```

The Reactive Streams Ingestion Library

The DB21c JDBC driver introduces a new Java library for high speed ingestion of massive volumes of data (sensors, Call Detail Records, logs) from large numbers of concurrent sources into the Oracle Database.
The library groups and buffers incoming data streams then gives control back to the client threads while using a separate thread pool for database I/O, using the Direct Path insert into database blocks, i.e., bypassing the SQL layer.



Java Program
1
Reactive Streams Ingestion Library
2
Record Grouping and Routing
3
Thread Scheduling
4
Direct Path Insert
UCP Pooling and Routing (Shard & Rac)

Figure 3 Reactive Streams Ingestion

The library (`rsi.jar`) furnishes the following simple APIs:

- Push Publisher: simplest usage, no backpressure.
- Flow Publisher: implementing Java Flow API (Publishers, Subscribers and Subscriptions).
- Record API for Object Relational Mapping

The following short video illustrates RSI in action. See the Oracle JDBC doc for more details and code fragments.

JDBC Reactive Extensions

The DB21c JDBC driver has been extended to support asynchronous database access with non-blocking network I/O. The extensions expose standard Java Flow Subscriber and Publisher types and APIs that can interoperate with the R2DBC API and the reactive streams libraries including: Reactor, RxJava, Akka, Vert.x, Spring, jOOQ, Querydsl, Kotysa and so on which furnish operators (map, reduce, filters), concurrency modeling, monitoring, and tracing.

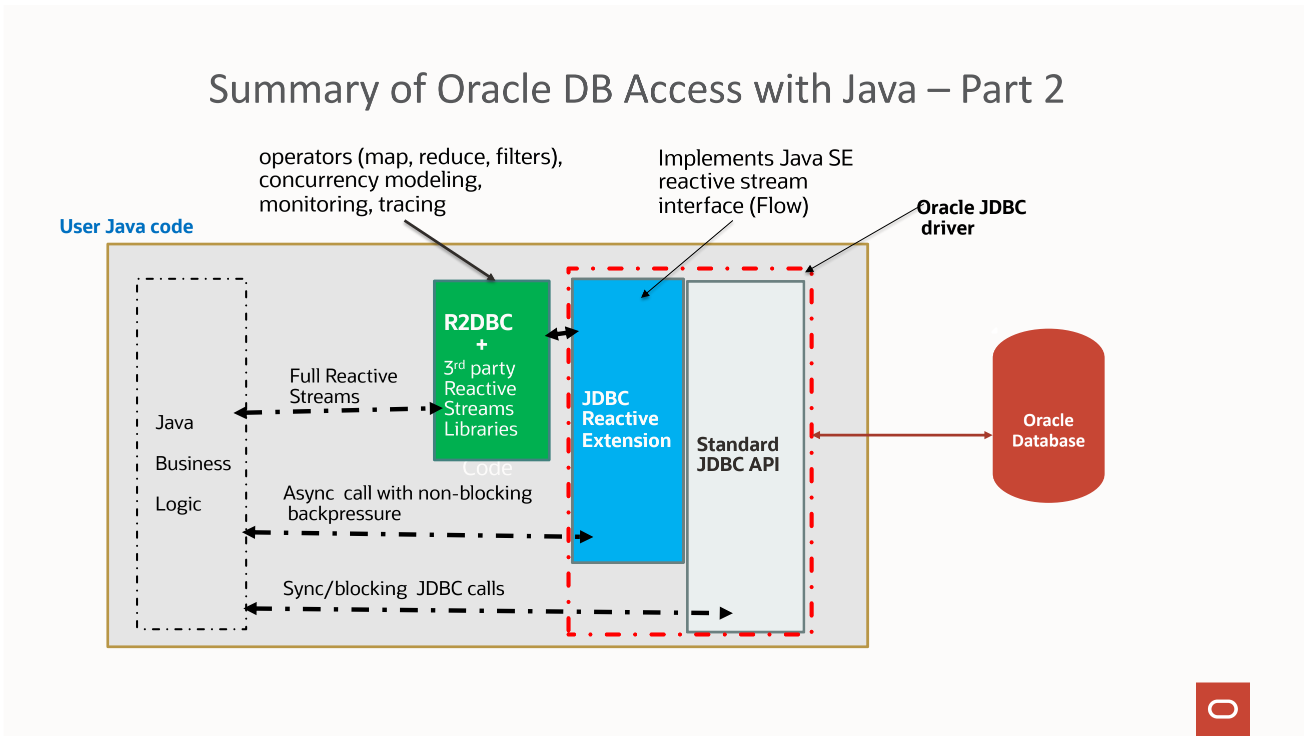

Summary of Oracle DB Access with Java – Part 2
operators (map, reduce, filters), concurrency modeling, monitoring, tracing
Implements Java SE reactive stream interface (Flow)
Oracle JDBC driver
User Java code
Java Business Logic
R2DBC + 3rd party Reactive Streams Libraries
Code
Full Reactive Streams
JDBC Reactive Extension
Standard JDBC API
Oracle Database
Async call with non-blocking backpressure
Sync/blocking JDBC calls

Figure 4 Database Access with Oracle JDBC 21c

The following code snippet shows how to open a connection, asynchronously.

```
/**
 * Asynchronously opens a new connection
 * @param dataSource Datasource configured with a URL, User, and Password
 * @return A Publisher that emits a single connection
 * @throws SQLException If a database access error occurs before the
 * connection can be opened
 */
Flow.Publisher<OracleConnection> openConnection(DataSource dataSource)
  throws SQLException {
  return dataSource.unwrap(OracleDataSource.class)
    .createConnectionBuilder()
    .buildConnectionPublisherOracle();
}
```

See the Oracle JDBC doc for more details.

Driver Configuration for GraalVM Native Image

GraalVM speeds up mission critical applications (faster startup, low runtime memory, and so on). The DB21c JDBC driver have been instrumented with GraalVM configuration in `META-INF/native-image`. Additional code in the driver relates to the orai18n and xmlparserv2 companion jars used respectively for NLS or Internalization support, and XML parsing.

The following steps allows running a plain JDBC code DataSourceSample.java as a native image.

1) Install Native Image with the GraalVM Updater tool

```
gu install native-image
```

2) Modify the DB_URL at line 40 to point to your database then compile the java file with the proper classpath:

```
javac -cp .:./ojdbc11.jar DataSourceSample.java
```

3) Run the Native Image builder

```
native-image -cp .:./ojdbc11.jar DataSourceSample
```

4) Run the image (the JVM is no longer needed at this stage):

```
./datasourcesample
```

In addition to instrumenting the driver, the JDBC dev team has been actively working with the Helidon, Micronaut, and Quarkus teams to integrate the driver and companion jars with their GraalVM native images.

Sharding Datasource

Java SE 9 JDBC 4.3 introduces the `ShardingKey` and the `ShardingKeyBuilder` interfaces for creating a ShardingKey however, existing applications must be modified to use these new APIs. |This is a problem for many Java developers.
The DB21c JDBC driver introduces a new JDBC datasource for simplifying Java connectivity to Sharded databases.
It removes the requirement to explicitly set the Sharding key before requesting a connection. The SQL statements intended for a single shard must contain the Sharding key in the `WHERE` clause.
Example: `select id, name from customer where id = ?`

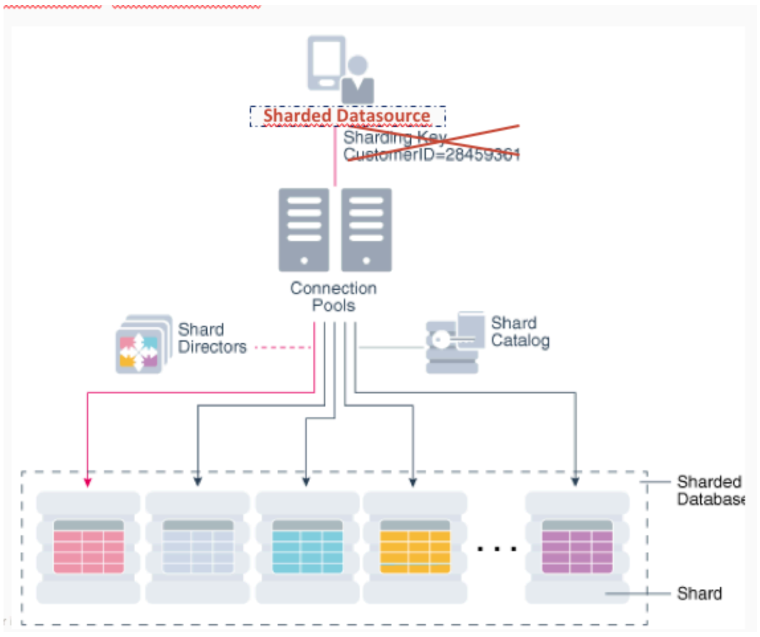


Figure 5 Sharding Datasource

Here are the requirements and best practices for using the Sharding datasource:

1) Java developers must set the `oracle.jdbc.useShardingDriverConnection` system property to true (in code or at the JDK level)

```
 Properties prop = new Properties();
prop.setProperty("oracle.jdbc.useShardingDriverConnection", "true");
```

2) Java developers must use the Shard director (a.k.a. GSM) connect string; it routes connections to the appropriate Shard

```
final static String gsmURL = "jdbc:oracle:thin:@(DESCRIPTION = (ADDRESS = (HOST =
...)(PORT = ...)(PROTOCOL = tcp))(CONNECT_DATA = (SERVICE_NAME = ...)))";
```

3) The Sharding datasource supports local transaction against a single shard when `Auto-COMMIT` is `OFF` however, the Java developer must set the datasource property `allowSingleShardTxn` to TRUE to indicate that local transactions always start and end on a single Shard.

The following are the known limitations as of DB21c

- The Sharding datasource supports only the type 4 JDBC Thin driver. It does not support the type 2 JDBC OCI driver or the RDBMS server-side type 2 driver (a.k.a. KPRB driver).
- The Sharding data source does not currently support transactions spanning multiple shards.
- The Sharding datasource does not support some of the Oracle JDBC extensions such as the Direct Path load, and JDBC Dynamic Monitoring Service (DMS) metrics.

The following video shows the Sharding datasource in action.

Zero Downtime for Mission Critical Deployment

Zero downtime is an essential *Quality of Service* for mission critical applications. Application Continuity (AC), Transparent AC (TAC) and Transaction Guard are major zero downtime mechanisms for client Java applications. Java code running in the database also benefit from the zero-brownout patching and security enhancements.

Application Continuity (AC) & Transparent AC

Application Continuity (AC) consists in recording, in driver memory, all database calls made during an application unit of work (typically a transaction) and replaying these calls— upon the failure of an instance, host or network — against another instance of the same database, after hiding the exception. Upon replay, if the outcome is identical (i.e., result set checksum) then AC is successful and the control is given back to the application to continue, as if nothing has happened; if not, the exception is re-cast and visible the Java code. I described AC and TAC from Java developer's perspective in the Zero-Downtime section of this blog post.

In this release, a new attribute named `RESET_STATE` of the database service has been added – this is transparent to your Java code -- cleans states set/used by the application when the unit of work i.e., transaction, completes. Specifically, cursors are cancelled, `PL/SQL globals` are cleared, temporary session tables are truncated, and temporary session lobs are cleared.

JVM in the Database - High-Availability and Security Enhancements

The JVM in the database (a.k.a. OJVM) is used for: in-place data processing; calling out Web-Services, Hadoop servers, 3rd party databases and legacy systems; running 3rd party Java libraries; running Java-based languages (e/g., Jython, Groovy Kotlin, Clojure, Scala, JRuby). OJVM is also used by database components such as AQ – JMS, XDB, Spatial, Scheduler, Java XA and OLAP.

This release furnishes the long-awaited zero brownout and rolling patching of OJVM in clustered database (RAC) environments and additional security enhancements.

RAC Rolling OJVM Patching

This release furnishes zero brownout and RAC rolling OJVM patching. RAC Rolling patching of OJVM was possible in the previous releases (18c and 19c) however, there was a few seconds brown-out period during which the Java is not available in the entire cluster.

Security Enhancements

Lockdown profile

A *lockdown profile* is a mechanism for restricting certain operations or functionalities in pluggable database (PDB) and the container database (CDB). In the previous releases, the following lockdown profiles were implemented: disable OS file access and networking to/from Java (in DB 19c release) and disabling granting `java.lang.RuntimePermission` and `java.io.FilePermission`.
In this release, the following lockdown profiles are implemented: confining OS file accesses in paths within `PATH_PREFIX`; the ability to specify OS user identity when forking OS process.

Native Network Encryption (NNE)

As part of the July 2021 Critical Patch Update (CPU), Oracle introduced several NNE changes to deal with vulnerability CVE-2021-2351 and prevent the use of weaker ciphers. Oracle JDBC Thin driver v21.3.0.0, v 19.12.0.0 and v 18.15.0.0 contain the NNE changes. These require server side changes as well. The recommendation is to use both client and server that contain the fix for NNE and disallow weak crypto algorithms.

In case you are using an older version of the drivers that does not contain a fix for NNE, make sure to patch the driver and set a new property `“oracle.net.allow_weak_crypto”` to “true” to allow NNE connectivity. Check out this property in `oracle.jdbc.OracleConnection` class.

Refer to MOS note 2791571.1 for more details.

Conclusion

This technical brief walked you through the new and recent enhancements in the Oracle database 21c that simplify the onboarding or the experience of Java developers and architects. These new enhancements also simplify the development and deployment of mission critical and Cloud native Java applications.

The reference section hereafter furnishes links to the developer guides and javadocs. Our landing page https://www.oracle.com/jdbc/ furnishes up to date resources.

References

JDBC Developer's Guide

Java Developer's Guide

Universal Connection Pool Developer's Guide

JDBC Java API Reference

Universal Connection Pool API Reference

RAC FAN Events Java API Reference

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 facebook.com/oracle twitter.com/oracle