40
Diagnosability in JDBC

The JDBC diagnosability feature is enhanced in this release to make it more user-friendly.

The enhancement also facilitates the use of the JDBC drivers in the Oracle Cloud Infrastructure (OCI) environment because unlike the On-Premises users, the Cloud users do not have absolute access to the system. The enhanced diagnosability feature provides diagnostic capabilities that are accessible and configurable in the Cloud environment.

40.1 Overview of JDBC Diagnosability

JDBC diagnosability feature is a client-only feature. Like in previous releases, it uses the Java Util Logging framework. It does not work with the Server-side Thin driver or the Server-side internal driver.

In the previous releases, the debug `JAR` files are used for logging purposes. These files are indicated with a `_g` in the file name, for example, `ojdbc8_g.jar` or `ojdbc11_g.jar`, and must be included in the `CLASSPATH`. The enhanced diagnosability feature in the Database 23ai Release, eliminates the need to use the debug `JAR` files, and also the need to switch between the regular `JAR` files and the debug `JAR` files. Now, the regular `JAR` files include logging capabilities, which can be turned on with the following property:

```
oracle.jdbc.diagnostic.enableLogging=true
```

This property can be set either through a Java System property (using the `-D` option) or through a `DataSource` property.

The logging configuration file can be configured in the following way:

```
-Djava.util.logging.config.file=./logging.config
```

For example, the following logging configuration file turns on the JDBC logging in the console with the `OracleSimpleFormatter`:

```
handlers = java.util.logging.ConsoleHandler
oracle.jdbc.level = ALL
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
```

Logging can be enabled dynamically through the `com.oracle.jdbc.diagnosability` JDBC MBean. This MBean defines multiple operations such as `enableLogging`, `disableLogging`, or `enableLoggingByConnectionIdPrefix`, and so on. All these operations can also be called programmatically by invoking the relevant operation. For example:

```
void enableLogging(boolean enable) {
  try {
    Object loader = oracle.jdbc.driver.OracleDriver.class.getClassLoader();
```

```
    String loaderName =
            (loader == null ? "nullLoader" : loader.getClass().getName());
    String name = loaderName + "@" +
                      Integer.toHexString(
                              (loader == null ? 0 : loader.hashCode())
                      // If the same class loader loads the JDBC drivers
multiple times, then each
                      // subsequent MBean increments the value of the
loader.hashCode() method, so as to
                      // create a unique name. It may be problematic to
identify which MBean is
                      // associated with which JDBC driver instance.
                      );
    javax.management.ObjectName diagnosticMBeanObjectName = new
javax.management.ObjectName("com.oracle.jdbc:type=diagnosability,name=" +
name);

    // get the MBean server
    javax.management.MBeanServer mbs =
java.lang.management.ManagementFactory.getPlatformMBeanServer();

    // find out if logging is enabled or not
    System.out.println("LoggingEnabled = " +
mbs.getAttribute(diagnosticMBeanObjectName, "LoggingEnabled"));

    // enable logging
    if(enable)
      mbs.invoke(diagnosticMBeanObjectName, "enableLogging", null, null);
    else
      mbs.invoke(diagnosticMBeanObjectName, "disableLogging", null, null);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Features of JDBC Diagnosability

The improved JDBC diagnosability feature is built upon the following features of Oracle Database:

- **Observability**: It enables Java developers to access summary information about the execution and performance of JDBC.
- **Diagnosability**: It records critical execution state in the event of a failure. Ideally, such a state is sufficient to diagnose the first occurrence of a failure and come up with a resolution of the problem causing the failure. You must balance the amount of state information recorded against the cost of recording such a state.
- **Execution Trace**: It records the execution sequence details. As execution trace has significant cost involved, you must enable it only in limited contexts.

Each diagnosability feature has two modes, public and sensitive. In the public mode, these features do not record or persist sensitive information. This severely limits the amount of information captured, and reduces the effectiveness of the diagnosability features. In the sensitive mode, these features record and persist sensitive information. You must be a privileged user for permitting the sensitive mode.

The sensitive mode is controlled by two switches, one for enabling and one for permitting. The sensitive mode for each of the diagnosability feature is permitted by setting the appropriate Java command-line switch. If the `permit` switch is not set on the Java command line as a Java System property, then the sensitive mode cannot be enabled. After permitting the sensitive mode, you must enable it, as shown in the following example:

```
-Doracle.jdbc.diagnostic.permitSensitiveDiagnostics=true
-Doracle.jdbc.diagnostic.enableSensitiveDiagnostics=true
```

Note:

- Permitting the sensitive mode does not automatically enable it. Attempting to enable sensitive mode, when not permitted, causes an irrecoverable error.
- Similarly to enabling logging, enabling the sensitive mode can be done dynamically through the MBean.
- When the sensitive mode is enabled, the SQL text appears in the logs, but not the parameter data.

Examples

The following code snippets show how to configure the logging configuration file:

Example 40-1 Configuring the Logging File

```
handlers = java.util.logging.ConsoleHandler
oracle.jdbc.level = FINEST
java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
```

Example 40-2 Configuring the Logging File (detailed)

```
handlers=java.util.logging.FileHandler
java.util.logging.FileHandler.level = FINEST
java.util.logging.FileHandler.formatter =
oracle.jdbc.diagnostics.OracleSimpleFormatter
java.util.logging.FileHandler.pattern = client.log
java.util.logging.FileHandler.limit = 1000000000
java.util.logging.FileHandler.count = 1000
oracle.ucp.level=FINEST
oracle.jdbc.level=FINEST
```

40.2 The Diagnose First Failure Feature

One of the major features of the enhanced diagnosability is the Diagnose First Failure feature.

When you enable this feature, it diagnoses the first occurrence of a failure, and the most critical trace information is captured in memory. As that memory fills, the oldest trace records are overwritten, and are subsequently dumped into `java.util.logging`, when signaled. One of the following can signal the dumping of the trace records:

- Occurrence of an error in the connection
- Client application code calling an API
- An MBean

This feature captures only the most critical information, that is, the extent of information that provides a reasonable chance of diagnosing the most likely problems. In the public mode, this information is severely restricted, so that any sensitive information is omitted. In the sensitive mode, this information includes the entire network conversation.

This feature is enabled by default. You can disable it either by setting the `CONNECTION_PROPERTY_ENABLE_DIAGNOSE_FIRST_FAILURE` property to `FALSE` or through the `DiagnosticMBeans` interface.

> **See Also:**
>
> Oracle Database JDBC Java API Reference