9

Post-Execution Debugging of MLE JavaScript Modules

The ability to easily debug code is central to a good developer experience. MLE provides the option to perform post-execution debugging on your JavaScript source code in addition to standard print debugging.

Post-execution debugging allows efficient collection of runtime state during program execution. Once execution of the code has completed, the collected data can be used to analyze program behavior and discover bugs that require attention. To perform post-execution debugging, you provide a debug specification that identifies the debugging information to be collected. A debug specification is a collection of debugpoints, each of which specify a location in the source code where debug information should be collected, as well as what information to collect. Debugpoints can be conditional or unconditional.

> **Note:**
>
> Post-execution debugging can only be applied to JavaScript code that is deployed as MLE modules. This debugging feature cannot currently be used when deploying code via dynamic execution.

> **Note:**
>
> MLE built-in modules such as the MLE JavaScript driver and MLE bindings cannot be debugged via post-execution debugging. An attempt to debug a built-in module will cause an `ORA-04162` error to be raised.
>
> For more information about MLE built-in modules, see Server-Side JavaScript API Documentation.

Module debugpoints apply to all executions of the module code, including via MLE call specifications, as well as via module import, whether from a dynamic MLE source or from another MLE module. Once enabled, a debug specification is active either until it is disabled or replaced by a new debug specification, or until the session ends.

Topics

- Specifying Debugpoints
  Debugpoints are specified using a JSON document encoded in the database character set.
- Managing Debugpoints
  Debugging can be enabled in a session by calling the procedure `dbms_mle.enable_debugging` with a debug specification.
- Analyzing Debug Output
  Output from debugpoints is stored in the Java Profiler Heap Dump version 1.0.2 format.

- Error Handling in MLE
  Errors encountered during the execution of MLE JavaScript code are reported as database errors.

Specifying Debugpoints

Debugpoints are specified using a JSON document encoded in the database character set.

Each debugpoint has the following elements:

- A ***location*** in the source code where the information is collected
- An ***action*** that describes what information to collect
- An optional ***condition*** that controls when debug information should be collected

Example 9-1 JSON Template for Specifying Debugpoints

```
{
at: <location-spec>,
action: [ <action-spec>, ... ],
[ condition: <condition-spec> ]
}
```

- Debugpoint Locations
  Debugpoint locations are specified via the line number in the source code of the application being debugged.
- Debugpoint Actions
  MLE post-execution debugging supports two kinds of actions: `watch` and `snapshot`.
- Debugpoint Conditions
  Both `watch` and `snapshot` can be controlled via conditions specified in the `condition` field.

Debugpoint Locations

Debugpoint locations are specified via the line number in the source code of the application being debugged.

The name of the MLE module to be debugged is specified via the `name` field and the location within the module where debug information is to be collected is specified via the `line` field. Example 9-4 provides an example JSON document with sample values.

Debugpoint Actions

MLE post-execution debugging supports two kinds of actions: `watch` and `snapshot`.

The `watch` action allows you to log the value of the variable named in the `id` field. The optional `depth` field provides you with control over the depth to which values of composite type variables are logged.

The `snapshot` action logs the stack trace at the point the `snapshot` action is invoked, along with the values of the local variables in each stack frame. A higher cost of performance is required by `snapshot` compared with `watch` but it provides a greater depth of information. As with the `watch` action, the optional `depth` field can be used to control the depth of logging for each variable. The `depth` parameter for the `snapshot` action applies to all variables captured by the action.

More precisely, the `depth` parameter controls how deeply you traverse the object tree in order to capture the value of a variable. For example, consider the following variable with nested objects:

```
let x = {
a: {
    val: 42
},
b: 3.14
};
```

If the `depth` field is defined as `2`, the object tree would be traversed and the value of the nested object `a` would be captured, which in this case is `42`. If `depth` is specified as `1`, the traversal would end at the first level, which would produce the following results:

```
x = {
"a": {
    "<unreachable>": true
};
"b": 3.14
}
```

The `framesLimit` field provides you with control over the number of stack frames to be logged. The default is to log all stack frames. `framesLimit` only applies to `snapshot`. Take, for example, a call hierarchy where `a()` calls `b()` and `b()` calls `c()`. If you take a snapshot in `c()`, `framesLimit=1` would only capture the bottom-most stack frame (in this case, `c()`), `framesLimit=2` would capture the bottom two (in this case, `c()` and `b()`), and so on.

Example 9-2 JSON Template for Specifying Watch Action

To watch a variable, `type` must be set to `watch`. The `id` parameter is used to identify the variable or variables to watch and must be provided as either a string or an array of strings. The `depth` parameter is optional and is defined by a number.

```
actions: [
  { type: "watch",
    id: <string[]> | <string>,
    [depth : <number>] }
]
```

Example 9-3 JSON Template for Specifying Snapshot Action

To use the `snapshot` action, the `type` parameter must be set to `snapshot`. The `framesLimit` and `depth` fields are optionally provided as numbers.

```
actions: [
  { type: "snapshot",
    [framesLimit: <number>],
    [depth : <number>] }
]
```

Debugpoint Conditions

Both `watch` and `snapshot` can be controlled via conditions specified in the `condition` field.

The expression is evaluated in the context of the application at the location specified in the debugpoint and the associated action is triggered only if the expression evaluates to `true`.

There are no restrictions on the type of expression that can be included in the condition field. You must ensure that evaluating any expressions does not alter the behavior of the program being debugged.

Example 9-4 Watching a Variable in an MLE Module

The following code specifies a debugpoint for a module, `myModule1`, with two associated actions. A `watch` action for variable `x` with the logging depth restricted to `3`, and a `watch` action for variable `y` with no restrictions on logging depth. The debugpoint also has an associated `condition` so that the debugpoint actions only trigger if the condition (`x.id>100`) is met.

```
{
    "at" : {
        "name" : "myModule1",
        "line" : 314
    },
    "actions" : [
        { "type": "watch", "id" : "x", "depth" : 3 },
        { "type": "watch", "id" : "y" }
    ],
    "condition" : "x.id > 100"
}
```

Managing Debugpoints

Debugging can be enabled in a session by calling the procedure `dbms_mle.enable_debugging` with a debug specification.

In addition to an array of debugpoints, specified via the `debugpoints` field, a debug specification includes a version identifier, specified via the `version` field. The `version` field must be set to the value `"1.0"`. Debug specifications can include debugpoints for multiple MLE modules.

> **Note:**
>
> Debug specifications require module names to be provided in the same case that they are stored in the dictionary. By default, module names are stored in uppercase unless the name is enclosed in double-quotation marks during module creation.

The procedure `dbms_mle.enable_debugging` also accepts a `BLOB sink` to which the debug output is written.

After the call to `dbms_mle.enable_debugging`, all debugpoints included in the debug specification are active. Every time one of the debugpoints is hit, the associated debug information is logged. The debug information is written out to the `BLOB sink` when control

passes from MLE back to PL/SQL at the latest but could be written out in part or in full before this point:

- For dynamic MLE evaluations, control passes from MLE to PL/SQL when the call to `dbms_mle.eval` returns.
- For MLE call specifications, control passes from MLE to PL/SQL when the call to the MLE call specification returns.

The installed debugpoints are active for all executions of the MLE modules regardless of which user's privileges the MLE code executes with.

Calling `dbms_mle.enable_debugging` again in the same session replaces the existing set of debugpoints. Debugpoints remain active until either the session ends or the user disables debugging explicitly by calling `dbms_mle.disable_debugging`.

Example 9-5 Enabling Debugging of an MLE Module

The debug specification in this example references the module `count_module`, created at the beginning of Example 6-4, and module `in_out_example_mod`, created in Example 6-6.

```
DECLARE
  debugspec json;
  sink blob;
BEGIN
  debugspec:= json('
    {
      "version": "1.0",
      "debugpoints": [
        {
          "at": {
            "name": "COUNT_MODULE",
            "line": 7
          },
          "actions": [
            { "type": "watch", "id": "myCounter", "depth": 1 }
            ],
          "condition": "myCounter > 0"
        },
        {
          "at": {
            "name": "IN_OUT_EXAMPLE_MOD",
            "line": 16
          },
          "actions": [
            { "type": "snapshot" }
          ],
        }
      ]
    }
  ');
  dbms_lob.createtemporary(sink, false);
  dbms_mle.enable_debugging(debugspec, sink);
  --run application to debug
END;
/
```

- Debugging Security Considerations
  Users must either own the MLE modules being debugged or have debugging privileges to it. This is necessary because the debugging feature allows you to observe runtime state of the MLE code.
- COLLECT DEBUG INFO Privilege for MLE Modules
  The `COLLECT DEBUG INFO` object privilege for MLE modules controls whether a user who does not own a module, but has `EXECUTE` privilege, can still perform debugging on said module.

Debugging Security Considerations

Users must either own the MLE modules being debugged or have debugging privileges to it. This is necessary because the debugging feature allows you to observe runtime state of the MLE code.

Additionally, because the `condition` field allows you to execute arbitrary code, this could potentially be used to alter the runtime behavior of the code being debugged. Concretely, you can use post-execution debugging on an MLE module if,

- You own the MLE module, or
- You have the `COLLECT DEBUG INFO` object privilege on the MLE module.

Privileges are checked every time code in an MLE module with one or more active debugpoints is executed. If you attempt to install debugpoints without the necessary privileges, an `ORA-04164` error will be raised.

If an `ORA-04164` is encountered, either

- The user who installed the debugpoints must be granted the `COLLECT DEBUG INFO` privilege on the module in question, or
- The debugpoints for the module must be disabled to continue executing code in the module in that session.

COLLECT DEBUG INFO Privilege for MLE Modules

The `COLLECT DEBUG INFO` object privilege for MLE modules controls whether a user who does not own a module, but has `EXECUTE` privilege, can still perform debugging on said module.

For instance, consider an MLE module, `ModuleA`, owned by user `W`. User `W` creates an invoker's rights call specification for a function in `ModuleA` and grants `EXECUTE` on this call specification on user `V`. For user `V` to have the ability to debug the code in `ModuleA` when calling this call specification, user `W` must also grant them the `COLLECT DEBUG INFO` privilege on `ModuleA`.

User `W` could use the following statement to grant user `V` the privilege to debug `ModuleA`:

```
GRANT COLLECT DEBUG INFO ON ModuleA TO V;
```

The `COLLECT DEBUG INFO` privilege can subsequently be revoked if needed:

```
REVOKE COLLECT DEBUG INFO ON ModuleA FROM V;
```

Analyzing Debug Output

Output from debugpoints is stored in the Java Profiler Heap Dump version 1.0.2 format.

Every time a debugpoint is hit during execution, the debug information is saved as a heap dump segment. Once execution finishes, you have two options to analyze the debug output:

- Use the textual representation of the debug information obtained via the `dbms_mle.parse_debug_output` function.
- Export the `BLOB sink` containing the debug output to an `hprof` file and use any of a number of existing developer tools to analyze the information.

Topics

Textual Representation of Debug Output

The function `dbms_mle.parse_debug_output` takes as input a `BLOB` containing the debug information in the heap dump format and returns a JSON representation of the debug information.

The output of `dbms_mle.parse_debug_output` is an array of `DebugPointData` objects. `DebugPointData` represents the debug information logged every time a debugpoint is hit and comprises of an array of `Frame` objects. Each `Frame` includes the location in source code where the information was collected (the `at` field) and the names and values of local variables logged at that location (the `values` field). Note that the keys of `Frame.values` are the names of the variables logged and the values are the values of those variables.

Example 9-6 demonstrates how you can specify a debugpoint in a sample JavaScript program and then use the function `dbms_mle.parse_debug_output` to produce a textual representation of the debug output.

Example 9-6 Obtain Textual Representation of Debug Output

The debugging shown later in this example is performed on the JavaScript function `fib` defined in the module `fibunacci_module`:

```
CREATE OR REPLACE MLE MODULE fibunacci_module
LANGUAGE JAVASCRIPT AS
export function fib( n ) {

    if (n < 0) {
        throw Error("must provide a positive number to fib()");
    }
    if (n < 2) {
        return n;
    } else {
```

```
        return fib(n-1) + fib(n-2);
    }
}
/

CREATE OR REPLACE FUNCTION fib( p_value number)
RETURN NUMBER
AS MLE MODULE fibunacci_module
SIGNATURE 'fib(number)';
/
```

A debugpoint is placed at line 9 and then the `DBMS_MLE.PARSE_DEBUG_OUTPUT` function is used to view the debug information:

```
SET SERVEROUTPUT ON;
DECLARE
    l_debugspec JSON;
    l_debugsink BLOB;
    l_debuginfo JSON;
    l_value     NUMBER;
BEGIN
    l_debugspec := JSON ('
    {
        version : "1.0",
        debugpoints : [
            {
                "at" : {
                    "name" : "FIBUNACCI_MODULE",
                    "line" : 9
                },
                "actions" : [
                    { "type" : "watch", "id" : "n" }
                ],
            },
        ]
    }
    ');
    -- create a temporary lob to store the raw
    -- debug output
    DBMS_LOB.CREATETEMPORARY( l_debugsink, false );

    DBMS_MLE.ENABLE_DEBUGGING( l_debugspec, l_debugsink );

    -- run the application code
    l_value := fib(4);

    DBMS_MLE.DISABLE_DEBUGGING;

    -- retrieve a textual representation of the debug
    -- output
    l_debuginfo := DBMS_MLE.PARSE_DEBUG_OUTPUT( l_debugsink );
    DBMS_OUTPUT.PUT_LINE(
        json_serialize(l_debuginfo pretty)
    );
```

```
END;
/
```

Result:


```
[
  [
    {
      "at": {
        "name": "USER1.FIBUNACCI_MODULE",
        "line": 9
      },
      "values": {
        "n": 4
      }
    }
  ],
  [
    {
      "at": {
        "name": "USER1.FIBUNACCI_MODULE",
        "line": 9
      },
      "values": {
        "n": 3
      }
    }
  ],
  [
    {
      "at": {
        "name": "USER1.FIBUNACCI_MODULE",
        "line": 9
      },
      "values": {
        "n": 2
      }
    }
  ],
  [
    {
      "at": {
        "name": "USER1.FIBUNACCI_MODULE",
        "line": 9
      },
      "values": {
        "n": 2
      }
    }
  ]
]
```

Analyzing Debug Output Using Developer Tools

As an alternative to analyzing the textual representation of debug output, you also have the option to utilize tools such as JDeveloper, NetBeans, and Oracle Database Actions.

Once execution has finished, you can use the tool of your choice to inspect the values of local variables or to inspect the graph of variables at each point in time.

Integration with new tools can be developed as needed (e.g., Chrome Dev Tools) and UIs can be designed that are tailored specifically to the MLE use case.

> **Note:**
>
> Oracle Database Actions supports MLE post-execution debugging starting with Oracle Database 23ai, Release Update 23.1.2.

> **See Also:**
>
> *Using Oracle SQL Developer Web* for more information about using Database Actions with MLE

Error Handling in MLE

Errors encountered during the execution of MLE JavaScript code are reported as database errors.

The database error raised depends on the type of error encountered. For example, syntax errors raise `ORA-04160` while runtime errors (e.g., uncaught exceptions) raise `ORA-04161`. The error message for each database error provides a brief description of the error encountered. Additionally, the `DBMS_MLE` PL/SQL package provides procedures to query the MLE JavaScript stack trace for the last error encountered in a dynamic MLE execution context or an MLE module in the current session.

The same security checks are made when calling `DBMS_MLE.get_ctx_error_stack()` as when calling `DBMS_MLE.eval()`. Thus, you cannot retrieve error stacks for MLE JavaScript code executing in dynamic MLE execution contexts created by other users.

`DBMS_MLE` provides a similar function, `DBMS_MLE.get_error_stack()`, to access the MLE JavaScript stack trace for application errors encountered during the execution of MLE modules. The function takes the module name and optionally the environment name as parameters, returning the stack trace for the most recent application error in a call specification based on the given arguments. If the module name or environment name is not a valid identifier, an `ORA-04170` error is raised.

With MLE modules, it is only possible to retrieve the error stack for the module contexts associated with the calling user. This restriction avoids potentially leaking sensitive information between users via the error stack. A natural consequence of this restriction is that you cannot retrieve stack traces for errors encountered when executing definer's rights MLE call specifications owned by other users.

Example 9-7 Throwing `ORA-04161` Error and Querying the Stack Trace

Executing the following code will throw an `ORA-04161` error:

```
CREATE OR REPLACE MLE MODULE catch_and_print_error_stack
LANGUAGE JAVASCRIPT AS

export function f(){
    g();
}

function g(){
    h();
}

function h(){
    throw Error("An error occurred in h()");
}
/

CREATE OR REPLACE PROCEDURE not_getting_entire_error_stack
AS MLE MODULE catch_and_print_error_stack
SIGNATURE 'f()';
/

BEGIN
    not_getting_entire_error_stack;
END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-04161: Error: An error occurred in h()
ORA-04171: at h (USER1.CATCHING_AND_PRINTING_ERROR_STACK:10:11)
ORA-06512: at "USER1.NOT_GETTING_THE_ENTIRE_ERROR_STACK", line 1
ORA-06512: at line 2
*/
```

You can query the stack trace for this error using the procedure `DBMS_MLE.get_error_stack()`:

```
CREATE OR REPLACE PACKAGE get_entire_error_stack_pkg AS

    PROCEDURE get_entire_error_stack;

END get_entire_error_stack_pkg;
/

CREATE OR REPLACE PACKAGE BODY get_entire_error_stack_pkg AS

    PROCEDURE print_stack_trace( p_frames IN DBMS_MLE.error_frames_t ) AS
    BEGIN
        FOR i in 1 .. p_frames.count LOOP
```

```
            DBMS_OUTPUT.PUT_LINE( p_frames(i).func || '(' ||
            p_frames(i).source || ':' || p_frames(i).line || ')');
        END LOOP;
    END print_stack_trace;

    PROCEDURE do_the_work
    AS MLE MODULE catch_and_print_error_stack
    SIGNATURE 'f()';

    PROCEDURE get_entire_error_stack AS
        l_frames DBMS_MLE.error_frames_t;
    BEGIN
        do_the_work;
    EXCEPTION
    WHEN OTHERS THEN
        l_frames := DBMS_MLE.get_error_stack(
            'CATCH_AND_PRINT_ERROR_STACK'
        );
        print_stack_trace(l_frames);
        raise;
    END;
END get_entire_error_stack_pkg;
/

BEGIN
    get_entire_error_stack_pkg.get_entire_error_stack;
END;
/
```

The preceding code prints out the MLE JavaScript exception stack trace before raising the original error:

```
h(USER1.CATCH_AND_PRINT_ERROR_STACK:10)
g(USER1.CATCH_AND_PRINT_ERROR_STACK:6)
f(USER1.CATCH_AND_PRINT_ERROR_STACK:2)
BEGIN
*
ERROR at line 1:
ORA-04161: Error: An error occurred in h()
ORA-06512: at "USER1.GET_ENTIRE_ERROR_STACK_PKG", line 25
ORA-04171: at h (USER1.CATCH_AND_PRINT_ERROR_STACK:10:11)
ORA-06512: at "USER1.GET_ENTIRE_ERROR_STACK_PKG", line 11
ORA-06512: at "USER1.GET_ENTIRE_ERROR_STACK_PKG", line 18
ORA-06512: at line 2
```

- Errors in Callouts
  Database errors raised during callouts to SQL and PL/SQL via the MLE SQL driver are automatically converted to JavaScript exceptions.
- Accessing stdout and stderr from JavaScript
  MLE provides functionality to access data written to standard output and error streams from JavaScript code.

Errors in Callouts

Database errors raised during callouts to SQL and PL/SQL via the MLE SQL driver are automatically converted to JavaScript exceptions.

For most database errors, JavaScript code can catch and handle these exceptions as usual. However, exceptions resulting from critical database errors cannot be caught. This includes:

- Internal database errors (`ORA-0600`)
- Fatal database errors (`ORA-0603`)
- Errors triggered due to resource limits being exceeded (`ORA-04036`)
- User interrupts (`ORA-01013`)
- System errors (`ORA-7445`)

Exceptions resulting from database errors that are either not caught or are re-signaled cause the original database error to be raised in addition to an MLE runtime error (`ORA-04161`). You can retrieve the JavaScript stack trace for such exceptions using `DBMS_MLE.get_error_stack()` just like with other runtime errors.

Accessing stdout and stderr from JavaScript

MLE provides functionality to access data written to standard output and error streams from JavaScript code.

Within a database session, these streams can be controlled individually for each database user, MLE module, and dynamic MLE context. In each case, a stream can be:

- Disabled,
- Redirected to `DBMS_OUTPUT`, or
- Redirected to a user provided CLOB

- Accessing stdout and stderr for MLE Modules
  The `DBMS_MLE` PL/SQL package provides the procedures `set_stdout()` and `set_stderr()` to control the standard output and error streams for each MLE module context.
- Accessing stdout and stderr for Dynamic MLE
  The procedures `DBMS_MLE.set_ctx_stdout()` and `DBMS_MLE.set_ctx_stderr()` are used to redirect `stdout` and `stderr` for dynamic MLE contexts.

Accessing stdout and stderr for MLE Modules

The `DBMS_MLE` PL/SQL package provides the procedures `set_stdout()` and `set_stderr()` to control the standard output and error streams for each MLE module context.

Alternatively, `stdout` can be redirected to `DBMS_OUTPUT` using the function `DBMS_MLE.set_stdout_to_dbms_output()`. The `DBMS_MLE` package provides an analogous function fore redirection `stderr`: `DBMS_MLE.set_stderr_to_dbms_output()`.

`stdout` and `stderr` can be disabled for a module at any time by calling `DBMS_MLE.disable_stdout()` and `DBMS_MLE.disable_stderr()` respectively.

By default, `stdout` and `stderr` are redirected to `DBMS_OUTPUT`.

Note that the `CURRENT_USER` from an MLE function exported by the given MLE module may change depending on the `CURRENT_USER` when the function was called and whether the function is invoker's rights or definer's rights. A call to `DBMS_MLE.set_stdout()` or `DBMS_MLE.set_stderr()` by a database user, say `user1`, only redirects the appropriate stream when code in the MLE module executes with the privileges of `user1`.

In other words, one database user cannot ordinarily control the behavior of `stdout` and `stderr` for execution of an MLE module's code on behalf of another user.

All of these procedures take a module name and optionally an environment name as first and second arguments. This identifies the execution context whose output should be redirected. Omitting the environment name targets contexts using the base environment. Additionally, `set_stdout` and `set_stderr` take a user-provided CLOB as the last argument, specifying where the output should be written to.

Example 9-8 Redirect stdout to CLOB and DBMS_OUTPUT for MLE Module

Consider the following JavaScript module:

```
CREATE OR REPLACE MLE MODULE hello_mod
LANGUAGE JAVASCRIPT AS
    export function hello() {
        console.log('Hello, World from MLE!');
    }
/
```

The following call specification makes the exported function `hello()` available for calling from PL/SQL code.

```
CREATE OR REPLACE PROCEDURE MLE_HELLO_PROC
AS MLE MODULE hello_mod SIGNATURE 'hello';
/
```

The code below redirects `stdout` for the module `hello_mod` to a `CLOB` that can be examined later:

```
SET SERVEROUTPUT ON;
DECLARE
    l_output_buffer CLOB;
BEGIN
    -- create a temporary LOB to hold the output
    DBMS_LOB.CREATETEMPORARY(l_output_buffer, false);

    -- redirect stdout to a CLOB
    DBMS_MLE.SET_STDOUT('HELLO_MOD', l_output_buffer);

    -- run the code
    MLE_HELLO_PROC();

    -- retrieve the output buffer
    DBMS_OUTPUT.PUT_LINE(l_output_buffer);
END;
/
```

Executing the above produces the following output:

```
Hello, World from MLE!
```

Alternatively, `stdout` can be redirected to `DBMS_OUTPUT` using the function `DBMS_MLE.SET_STDOUT_TO_DBMS_OUTPUT()`:

```
SET SERVEROUTPUT ON;
BEGIN
    DBMS_MLE.SET_STDOUT_TO_DBMS_OUTPUT('HELLO_MOD');
    MLE_HELLO_PROC();
END;
/
```

This produces the same output as before:

```
Hello, World from MLE!
```

Accessing stdout and stderr for Dynamic MLE

The procedures `DBMS_MLE.set_ctx_stdout()` and `DBMS_MLE.set_ctx_stderr()` are used to redirect `stdout` and `stderr` for dynamic MLE contexts.

The `DBMS_MLE` package similarly provides the procedures `set_ctx_stdout_to_dbms_output()` and `set_ctx_stderr_to_dbms_output()` to redirect `stdout` and `stderr` for dynamic MLE contexts to `DBMS_OUTPUT`.

A call to one of these functions redirects the appropriate stream for all dynamic MLE code executing within the context. However, any calls to MLE functions via the MLE SQL driver use the redirection effect for the MLE module that implement the function.

Example 9-9 Redirect `stdout` to `CLOB` and `DBMS_OUTPUT` for Dynamic MLE

```
SET SERVEROUTPUT ON;
DECLARE
    l_ctx DBMS_MLE.context_handle_t;
    l_snippet CLOB;
    l_output_buffer CLOB;
BEGIN
    -- allocate the execution context and the output buffer
    l_ctx := DBMS_MLE.create_context();
    DBMS_LOB.CREATETEMPORARY(l_output_buffer, false);

    -- redirect stdout to a CLOB
    DBMS_MLE.SET_CTX_STDOUT(l_ctx, l_output_buffer);

    -- a bit of JavaScript code printing to the console
    l_snippet := 'console.log( "Hello, World from dynamic MLE!" )';

    -- execute the code snippet
    DBMS_MLE.eval(l_ctx, 'JAVASCRIPT', l_snippet);

    -- drop the execution context and print the output
    DBMS_MLE.drop_context(l_ctx);
```

```
    DBMS_OUTPUT.PUT_LINE(l_output_buffer);
END;
/
```

This produces the following output:

```
Hello, World from dynamic MLE!
```