4

Overview of Dynamic MLE Execution

Dynamic MLE execution allows developers to invoke JavaScript snippets via the `DBMS_MLE` package without storing the JavaScript code in the database.

As an alternative to executing JavaScript code using modules, MLE provides the option of dynamic execution. With dynamic execution, no JavaScript code is stored in the data dictionary. Instead, you can invoke snippets of JavaScript code via the `DBMS_MLE` package.

> **See Also:**
>
> - Server-Side JavaScript API Documentation for information about built-in module `mle-js-bindings`, used to exchange values between PL/SQL and JavaScript
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_MLE` package

Topics

- About Dynamic JavaScript Execution
  Developers can run JavaScript dynamically either inline or by loading files via `DBMS_MLE`. Dynamic MLE execution provides an additional method for using JavaScript to interact with the Oracle Database, as an alternative to using MLE modules.
- Dynamic Execution Workflow
  The steps required to perform dynamic MLE execution are described.
- Returning the Result of the Last Execution
  Use the `result` argument to get the outcome of the last execution.

About Dynamic JavaScript Execution

Developers can run JavaScript dynamically either inline or by loading files via `DBMS_MLE`. Dynamic MLE execution provides an additional method for using JavaScript to interact with the Oracle Database, as an alternative to using MLE modules.

The `DBMS_MLE` package allows users to execute JavaScript code inside the Oracle Database and seamlessly exchange data between PL/SQL and JavaScript. The JavaScript code itself can execute PL/SQL through built-in JavaScript modules. JavaScript data types are automatically mapped to Oracle Database data types and vice versa.

Developers can provide JavaScript code either as the value of a `VARCHAR2` variable or, in case of larger amounts of code, as a Character Large Object (CLOB). The JavaScript code is passed to the `DBMS_MLE` package where it is evaluated and executed.

Considering that `DBMS_MLE` is a PL/SQL package, there is mix of JavaScript and PL/SQL when dynamically executing code using `DBMS_MLE`, for example, in the following cases:

- Setup tasks such as providing the JavaScript code require an interaction with the PL/SQL layer.

- JavaScript code is executed by calling a function in `DBMS_MLE`.
- After JavaScript code completes execution, any errors that have been encountered are passed back to PL/SQL.

Dynamic Execution Workflow

The steps required to perform dynamic MLE execution are described.

Before a user can create and execute JavaScript code using `DBMS_MLE`, several privileges must be granted. For information about required privileges, see System and Object Privileges Required for Working with JavaScript in MLE.

The execution workflow for JavaScript code using `DBMS_MLE` is as follows:

1. Create an execution context
2. Provide JavaScript code either using a `VARCHAR2` or CLOB variable
3. Execute the code, optionally passing variables between the PL/SQL and MLE engines
4. Close the execution context

As with any code, it is considered an industry best practice to deal with unexpected conditions. You can do this in the JavaScript code itself using standard JavaScript exception handling features or in PL/SQL.

Topics

- Providing JavaScript Code Inline
  Using a quoting operator is the favored method for providing JavaScript code inline when performing dynamic execution.
- Loading JavaScript Code from Files
  The method for using a `BFILE` operator to read in a CLOB is described.

Providing JavaScript Code Inline

Using a quoting operator is the favored method for providing JavaScript code inline when performing dynamic execution.

A quoting operator, commonly referred to as a q-quote operator, is one option you can use to load JavaScript code by embedding it directly within a PL/SQL block. The use of this alternative quoting operator is suggested as the preferred method to provide JavaScript code inline with PL/SQL code whenever possible.

Note that while the q-quote operator is the recommended method for dynamic execution, delimiters such as `{{...}}` are used to enclose JavaScript code when using inline call specifications. To learn more about these delimiter options, see Creating an Inline MLE Call Specification.

Example 4-1 Using the Q-Quote Operator to Provide JavaScript Code Inline with PL/SQL

```
DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
BEGIN
    l_ctx := dbms_mle.create_context();
    l_snippet := q'~
```

```
// the q-quote operator allows for much more readable code
console.log(`The use of the q-quote operator`);
console.log(`greatly simplifies provision of code inline`);
~';
    dbms_mle.eval(l_ctx, 'JAVASCRIPT', l_snippet);
    dbms_mle.drop_context(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
END;
/
```

Result:

```
The use of the q-quote operator
greatly simplifies provision of code inline
```

Loading JavaScript Code from Files

The method for using a `BFILE` operator to read in a CLOB is described.

If you plan to use a linter to conduct code analysis, providing JavaScript code in line with PL/SQL may not be your best option for dynamic execution. Another method for providing JavaScript code is to read a CLOB by means of a `BFILE` operator. This way PL/SQL and JavaScript code can be cleanly separated.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about Large Objects

Example 4-2 Loading JavaScript code from a BFILE with DBMS_LOB.LOADCLOBFROMFILE()

This example illustrates the use of a `BFILE` and `DBMS_LOB.LOADCLOBFROMFILE()`.

The example assumes that you have read access to a directory named `SRC_CODE_DIR`. The source code file `hello_source.js` resides in that directory. Its contents are as follows:

```
console.log('hello from hello_source');
```

```
DECLARE
    l_ctx          dbms_mle.context_handle_t;
    l_js           CLOB;
    l_srcode_file  BFILE;
    l_dest_offset  INTEGER := 1;
    l_src_offset   INTEGER := 1;
    l_csid         INTEGER := dbms_lob.default_csid;
    l_lang_context INTEGER := dbms_lob.default_lang_ctx;
    l_warn         INTEGER := 0;
```

```
BEGIN
    l_ctx := dbms_mle.create_context();

    dbms_lob.createtemporary(lob_loc => l_js, cache => false);

    l_srcode_file := bfilename('SRC_CODE_DIR', 'hello_source.js');

    IF ( dbms_lob.fileexists(file_loc => l_srcode_file) = 1 ) THEN
        dbms_lob.fileopen(file_loc => l_srcode_file);
        dbms_lob.loadclobfromfile(
                dest_lob     => l_js,
                src_bfile    => l_srcode_file,
                amount       => dbms_lob.getlength(l_srcode_file),
                dest_offset  => l_dest_offset,
                src_offset   => l_src_offset,
                bfile_csid   => l_csid,
                lang_context => l_lang_context,
                warning      => l_warn
        );
        IF l_warn = dbms_lob.warn_inconvertible_char THEN
            raise_application_error(
                -20001,
                'the input file contained inconvertible characters'
            );
        END IF;

        dbms_lob.fileclose(l_srcode_file);
        dbms_mle.eval(
            context_handle => l_ctx,
            language_id    => 'JAVASCRIPT',
            source         => l_js
        );

        dbms_mle.drop_context(l_ctx);
    ELSE
        raise_application_error(
            -20001,
            'The input file does not exist'
        );
    END IF;

EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
END;
/
```

Result:

```
hello from hello_source
```

In some cases, you may need to mix dynamic MLE execution as shown in with MLE modules persisted in the database, as shown in Example 4-3.

Example 4-3 Loading JavaScript Code from a BFILE by Referencing an MLE Module from DBMS_MLE

The code for the JavaScript module is again stored in a file, as seen in Example 4-2. The example assumes that you have read access to a directory named `SRC_CODE_DIR` and the file name is `greeting_source.js`:

```
export function greeting(){
    return 'hello from greeting_source';
}
```

This example begins by creating an MLE module from `BFILE` using the contents of the preceding file. Before the module can be used by `DBMS_MLE`, an environment must be created first, allowing the dynamic portion of the JavaScript code to reference the module.

Dynamic MLE execution does not allow the use of the ECMAScript `import` keyword. MLE modules must instead be dynamically imported using the async/await interface shown in this example.

```
CREATE OR REPLACE MLE MODULE greet_mod
LANGUAGE JAVASCRIPT
USING BFILE(SRC_CODE_DIR, 'greeting_source.js');
/

CREATE OR REPLACE MLE ENV greet_mod_env
imports ('greet_mod' module greet_mod);

DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
BEGIN
    l_ctx := dbms_mle.create_context(
        environment => 'GREET_MOD_ENV'
    );
    l_snippet := q'~
(async () => {
    let { greeting } = await import('greet_mod');
    const message = greeting();
    console.log(message);
})();
~';
    dbms_mle.eval(
                l_ctx,
                'JAVASCRIPT',
                l_snippet
    );
    dbms_mle.drop_context(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
END;
/
```

Result:

```
hello from greeting_source
```

> **See Also:**
>
> Additional Options for Providing JavaScript Code to MLE for information about using `BFILE`s with MLE modules to load JavaScript code

Returning the Result of the Last Execution

Use the `result` argument to get the outcome of the last execution.

A variant of the `DBMS_MLE.eval()` procedure takes an additional CLOB argument, `result`. Such a call to `DBMS_MLE.eval()` appends the outcome of the execution of the last statement in the provided dynamic MLE snippet to the CLOB provided as the `result` parameter.

This option is useful in the implementation of an interactive application, such as a Read-Eval-Print-Loop (REPL) server, to mimic the behavior of a similar REPL session in Node.js.

Example 4-4 Returning the Result of the Last Execution

```
DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
    l_result  CLOB;
BEGIN
    dbms_lob.createtemporary(
                lob_loc => l_result,
                cache   => false,
                dur     => dbms_lob.session
    );

    l_ctx := dbms_mle.create_context();
    l_snippet := q'~
let i = 21;
i *= 2;
~';
    dbms_mle.eval(
                context_handle => l_ctx,
                language_id    => 'JAVASCRIPT',
                source         => l_snippet,
                result         => l_result
    );

    dbms_output.put_line('result: ' || l_result);
    dbms_mle.drop_context(l_ctx);
EXCEPTION
    WHEN OTHERS THEN
        dbms_mle.drop_context(l_ctx);
        RAISE;
```

```
END;
/
```

Result:

```
result: 42
```