2

Introduction to Oracle Database Multilingual Engine for JavaScript

Oracle Database supports a rich set of languages for writing user-defined functions and stored procedures, including PL/SQL, Java, and C. With Oracle Database Multilingual Engine (MLE), developers have the option to run JavaScript code through dynamic execution or with persistent MLE modules stored directly in the database.

The landscape of programming languages is rapidly evolving, with more developers choosing to use modern dynamic languages like JavaScript. Besides simpler syntax and support for modern language features, a key factor in the popularity of these languages is the existence of a rich module ecosystem. Developers often choose to use different languages to implement different parts of a project, based on the availability of suitable modules for the given task.

Whether or not a new language reaches widespread adoption frequently depends on community involvement. Once a language reaches some threshold of popularity, its ecosystem often starts expanding rapidly, attracting more and more developers. Many times, a rich set of features, libraries, and reusable code modules are created to support more widespread use.

The Oracle Database is renowned for its support of a rich ecosystem of programming languages. The most common programmatic server-side interface to the Oracle Database is PL/SQL. By using PL/SQL it is possible to keep business logic and data together, oftentimes offering significant improvements to efficiency in addition to providing a unified processing pattern for data, regardless of the client interface in use. With MLE, you can utilize PL/SQL to implement JavaScript modules, offering an additional avenue to interact directly with the database.

> **See Also:**
>
> *Oracle Database Development Guide* for more information about the programming languages supported by the Oracle database.

Topics

The Need for a Multilingual Engine

The benefits of using MLE to process data within the database are described.

When developers implement a *Smart-DB approach*, application logic and data coexist in the same database. Applying this strategy, the database is used as a full-fledged processing engine as opposed to simply a persistence layer or a simple REST API. Making use of the database for processing data where it lives can provide numerous advantages in the form of enhanced security, potential elimination of network round-trips, and better data quality thanks to the use of referential integrity.

The database's optimizer also benefits from this approach. Using referential integrity constraints allows it to know more about the data it's working with. Performance benefits can also be realized when using set-based SQL and oftentimes, database servers are more powerful than the machines serving the application's front-end, further speeding up processing time.

The Smart-DB approach requires you to be familiar with the programming languages offered by the database system to make the best use of the concept. The only other option is to use a client-side driver to extract data from the database to a middleware system or client machine for processing.

With the ever-increasing data volumes to be handled, especially for batch-processing, transferring large quantities of data from the database to a client can be problematic for the following reasons:

- The transfer of database information between servers is time consuming and can cause significant network overhead
- Latencies are often significantly increased; the cumulative effect can be very noticeable, especially for "chatty" applications
- Processing large data volumes in a middle-tier or client requires these environments to be equipped with large amounts of DRAM and storage, adding cost
- Data transfer between machines, especially in cloud environments, is often subject to regulatory control due to the inherent security risks and data protection requirements

Processing data *within* the database is a common strategy for mitigating against many of these problems.

With the introduction of Oracle Database Multilingual Engine (MLE), JavaScript is added to the database. The inclusion of JavaScript acknowledges the language's popularity and opens its extensive ecosystem for server-side database development.

With MLE, you can use idioms and tools available in JavaScript's ecosystem, as well as deploy and use modules from popular repositories such as Node Package Manager (NPM) right in the database. Furthermore, you can move between application tiers, providing more flexibility to teams dealing with varying workloads. The large pool of JavaScript talent can help staff existing and upcoming projects.

Overview of JavaScript

One of the most popular programming languages today, JavaScript runs on any machine with a JavaScript engine. Developers prefer JavaScript mainly for the ease of scripting to develop end-to-end applications and for fast execution.

JavaScript (JS) has come a long way since its inception as a browser-based solution for interactive web pages. While its popularity for front-end development remains strong, it has found its way into back-end development as well. For example, Node.js and Deno are very popular in that space.

At its core, JavaScript is an interpreted language with support for many modern programming styles. JavaScript is continually enhanced by a governing body known as ECMA International with new standards released annually.

JavaScript features both a functional as well as an object oriented interface. Despite the name, JavaScript is very different from Java, although its syntax intentionally mimics many constructs known in other popular languages. The learning curve is eased by providing a familiar looking syntax.

Soft factors such as a very large and active community as well as the language's rich set of libraries make it an attractive choice for development.

With the introduction of Oracle Database Multilingual Engine (MLE), it is possible to execute JavaScript directly in the Oracle database. Data-intensive applications can benefit from moving processing logic from the middle-tier to the database.

> **See Also:**
>
> Developer.mozilla.org for more information about JavaScript

Overview of Multilingual Engine for JavaScript

MLE allows you to run and store JavaScript directly in the Oracle Database.

Using MLE enables users of the Oracle Database to run the following, written in JavaScript:

- Stored procedures
- Stored functions
- Code in a PL/SQL package namespace
- Anonymous, dynamic code snippets (in a way that is similar to `DBMS_SQL`)

MLE is supported when connecting to the database using a dedicated server connection on Linux x86-64 or Linux for Arm (aarch64). Certain data types are not supported, listed in full at Unsupported Data Types.

> **Note:**
>
> Shared server connections and those using Database Resident Connection Pool (DRCP) cannot make use of MLE.

Topics

- JavaScript Implementation Details
  The MLE implementation of JavaScript is compliant with ECMAScript 2023.

- Invoking JavaScript in the Database
  JavaScript can be invoked through dynamic execution or through call specifications, which either reference MLE modules or inline JavaScript functions.
- Introduction to Dynamic Execution
  Anonymous JavaScript code snippets can be executed via the `DBMS_MLE` PL/SQL package.
- Introduction to MLE Module Calls
  It is possible to create JavaScript modules as schema objects that are stored persistently in the database.
- About MLE Execution Contexts
  An MLE execution context is a standalone, isolated runtime environment, designed to contain all runtime state associated with the execution of JavaScript code. Runtime state includes global variables as well as the state of the language environment.
- About Restricted Execution Contexts
  The `PURE` keyword can be specified on MLE environments and JavaScript inline call specifications to create restricted JavaScript execution contexts.

JavaScript Implementation Details

The MLE implementation of JavaScript is compliant with ECMAScript 2023.

Adhering to the ECMA standard, the JavaScript implementation as found in MLE is consciously created as a *pure implementation*. Native JavaScript network and file I/O operations are not supported in the same way that they are in Node.js and Deno for security reasons. The use of network and file I/O is possible with MLE, however, you must employ PL/SQL APIs such as `UTL_HTTP` and `UTL_FILE`.

The WEB API, Fetch, is not available by default in the global space but can be enabled by importing `mle-js-fetch`.

Objects not included in the ECMA standard, including common objects used in front-end code such as the Window object, are also not available with MLE. Nevertheless, MLE does provide easy and efficient access to SQL, which is able to execute close to the data. Console output is passed to `DBMS_OUTPUT` by default but can be redirected and stored in a user provided CLOB if required.

Users require specific privileges before they can interact with MLE. These can broadly be classified into:

- Permission to use MLE and run JavaScript code
- Execute dynamic JavaScript in the database
- Create JavaScript modules and externalize them via PL/SQL code

The database engine throws an error if you lack sufficient privileges required for the use of JavaScript.

> **See Also:**
>
> System and Object Privileges Required for Working with JavaScript in MLE for more information about privileges

Invoking JavaScript in the Database

JavaScript can be invoked through dynamic execution or through call specifications, which either reference MLE modules or inline JavaScript functions.

Generally speaking, server-side JavaScript code can be invoked in two ways:

- Dynamically via the `DBMS_MLE` package
- Using PL/SQL code referencing functions exported in JavaScript modules (so-called MLE module calls) or functions defined directly in the DDL

Regardless of which of the two methods is used, all JavaScript code runs in an execution context. Its purpose is to encapsulate all runtime state associated with the processing of JavaScript code. The MLE execution context corresponds to the ECMAScript execution context for JavaScript.

Before you can execute any JavaScript in the database, you must ensure that MLE is not disabled for your session, PDB, or CDB. For information about how to confirm this, see MLE_PROG_LANGUAGES Initialization Parameter. In order to take full advantage of MLE, you must have necessary privileges to execute the JavaScript language, execute dynamic MLE, create MLE schema objects, and so on.

See Also:

- System and Object Privileges Required for Working with JavaScript in MLE
- Ecma-international.org for more information about the ECMAScript execution context

Introduction to Dynamic Execution

Anonymous JavaScript code snippets can be executed via the `DBMS_MLE` PL/SQL package.

The procedure `DBMS_MLE.eval()` is used to execute dynamic MLE snippets. The procedure takes the following arguments:

Argument Name	Type	Optional?
CONTEXT_HANDLE	RAW(16)	N
LANGUAGE_ID	VARCHAR2(64)	N
SOURCE	CLOB	N
RESULT	CLOB	Y
SOURCE_NAME	VARCHAR2	Y

The argument `SOURCE_NAME` is optionally used to provide a name for the otherwise randomly-named JavaScript code block.

JavaScript code can be provided inline with PL/SQL as shown in the following code:

```
SET SERVEROUTPUT ON;
```

```
DECLARE
    l_ctx DBMS_MLE.context_handle_t;
    l_jscode CLOB;
BEGIN
    l_ctx := DBMS_MLE.create_context;
    l_jscode := q'~
        console.log('Hello World, this is DBMS_MLE')
    ~';
    DBMS_MLE.eval(
        context_handle => l_ctx,
        language_id => 'JAVASCRIPT',
        source => l_jscode,
        source_name => 'My JS Snippet'
    );
END;
/
```

Executing this example will result in the following being printed:

```
Hello World, this is DBMS_MLE
```

The code provided above demonstrates the following concepts of invoking JavaScript code dynamically:

- An execution context must be explicitly created
- JavaScript code is provided as a Character Large Object (CLOB) or `VARCHAR2` variable
- The context must be explicitly evaluated

Both PL/SQL and JavaScript are present when you execute JavaScript dynamically. The code snippets provided are not reusable outside of their namespace. The output of the call to `console.log` is passed to `DBMS_OUTPUT` for printing on the screen.

> **See Also:**
>
> - Overview of Dynamic MLE Execution for more details about dynamic execution with MLE
> - Returning the Result of the Last Execution for more information about the `RESULT` argument of the procedure `DBMS_MLE.eval()`

Introduction to MLE Module Calls

It is possible to create JavaScript modules as schema objects that are stored persistently in the database.

Once a JavaScript module has been defined, it can be used in SQL and PL/SQL as shown below:

```
CREATE OR REPLACE MLE MODULE helloWorld_module
LANGUAGE JAVASCRIPT AS
function helloWorld() {
```

```
    console.log('Hello World, this is a JS module');
}
export { helloWorld }
/
```

Before the exported JavaScript function can be invoked, a call specification must be defined. The code snippet below shows how to create a call specification for the JavaScript `helloWorld()` function in PL/SQL:

```
CREATE OR REPLACE PROCEDURE helloWorld_proc
AS MLE MODULE helloWorld_module
SIGNATURE 'helloWorld()';
/
```

The call specification, referred to as an MLE module call, publishes the JavaScript function `helloWorld()`. It can then be used just like any other PL/SQL procedure. The following snippet shows how to invoke the function along with the results:

```
SET SERVEROUTPUT ON

BEGIN
    helloWorld_proc;
END;
/
```

Result:

```
Hello World, this is a JS module
```

In addition to custom-built JavaScript modules as shown in the provided code, it is possible to load third-party JavaScript modules into the database. Note that Oracle recommends performing a security screening of third-party code according to industry best practice.

> **See Also:**
>
> - MLE JavaScript Modules and Environments for details about MLE modules and environments
> - MLE Security for more information about MLE security features and recommendations

About MLE Execution Contexts

An MLE execution context is a standalone, isolated runtime environment, designed to contain all runtime state associated with the execution of JavaScript code. Runtime state includes global variables as well as the state of the language environment.

> **Note:**
>
> An MLE execution context corresponds to an ECMAScript Execution Context for JavaScript.

MLE uses execution contexts in two different scenarios:

- With dynamic MLE execution, where you can create and use dynamic MLE contexts explicitly
- For calls from SQL and PL/SQL to functions exported by an MLE module

Dynamic Execution

Properties of dynamic MLE contexts are determined by the environment used at the moment the execution context is created. You have explicit control over which execution context is used for each dynamic MLE snippet, with each execution context running code on behalf of a single user.

There is no limit to how many dynamic MLE execution contexts can be created in a session, or how they are shared across different code snippets. Code snippets in JavaScript share all global variables with other code snippets running in the same execution context.

MLE Modules

Contexts for MLE module calls from SQL or PL/SQL are created implicitly on demand. Here, the properties are determined by the MLE environment referenced in the call specification at the moment of context creation. The environment can be used to specify language options and to make MLE modules available for import.

MLE modules never share an execution context with other modules or dynamic MLE snippets. Additionally, separate execution contexts are used when code from the same MLE module is executed on behalf of different users. MLE creates a dedicated execution context for each combination of MLE module and environment. Two call specifications that specify either different modules or different environments are executed in separate module contexts.

> **See Also:**
>
> - Specifying Environments for MLE Modules for more information about MLE environments
> - Execution Contexts for information about how execution contexts are used to enforce runtime state isolation

About Restricted Execution Contexts

The `PURE` keyword can be specified on MLE environments and JavaScript inline call specifications to create restricted JavaScript execution contexts.

In-database JavaScript code can leverage database functionality, such as SQL execution, using APIs like the MLE JavaScript SQL Driver and SODA. PURE execution disallows access to stateful database APIs inside JavaScript, meaning the execution is completely unprivileged. In a PURE environment, JavaScript code cannot read or write any database state, such as tables, procedures, and objects.

The only possible interaction with the database during PURE execution is through inputs and outputs to JavaScript code. This can be in the form of data provided to MLE from the database through user-defined function arguments for call specifications, as well as symbols exported using `DBMS_MLE.EXPORT_TO_MLE`. Reference types, such as LOBs passed to MLE, can be accessed (read or written) during PURE execution. Additionally, PURE execution does not restrict access to supported data types.

In many situations, JavaScript user-defined functions are purely computational and don't require access to powerful APIs such as the MLE JavaScript SQL driver or the Foreign Function Interface (FFI). PURE execution serves as a method to isolate certain code, such as third-party JavaScript libraries, from the database itself. This isolation can reduce the attack surface of supply chain attacks, in which access to the database state is a security concern. Using PURE execution also allows less-privileged developers to create these restricted user-defined functions without requiring additional access or privileges to the database state or network.

The following JavaScript APIs and global classes and functions are not available during PURE execution:

- JavaScript APIs:
  - `mle-js-oracledb`
  - `mle-js-plsql-ffi`
  - `mle-js-fetch`
- Global classes and functions:
  - `session`
  - `soda`
  - `plsffi`
  - `oracledb`
  - `require`

JavaScript APIs that do not interact with database state, such as `mle-js-plsqltypes` and `mle-js-encodings` remain accessible during PURE execution.

The `PURE` keyword can be specified in inline call specifications, in module call specifications, and using `DBMS_MLE`. The following are examples of the syntax in each case:

- Module call specification:

```
CREATE OR REPLACE MLE MODULE pure_mod
LANGUAGE JAVASCRIPT AS
export function helloWorld() {
```

```
    console.log('Hello World, this is a JS module');
}
/

CREATE OR REPLACE MLE ENV pure_env
IMPORTS( 'pure_mod' MODULE pure_mod) PURE;

CREATE OR REPLACE PROCEDURE helloWorld
AS MLE MODULE pure_mod ENV pure_env SIGNATURE 'helloWorld';
/
```

- Inline call specification:

```
CREATE OR REPLACE PROCEDURE helloWorld
AS MLE LANGUAGE JAVASCRIPT PURE
{{
    console.log('Hello World, this is a JS inlined call specification');
}};
/
```

- Using `DBMS_MLE`:

```
SET SERVEROUTPUT ON;
DECLARE
    l_ctx     dbms_mle.context_handle_t;
    l_snippet CLOB;
BEGIN
    -- to specify PURE execution with DBMS_MLE, use an environment
    -- that has been created with the PURE keyword
    l_ctx := dbms_mle.create_context(environment => 'PURE_ENV');
    l_snippet := q'~
        console.log('Hello World, this is dynamic MLE execution');
    ~';
    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;
/
```

Introduction to Debugging JavaScript Code

MLE allows you to debug your JavaScript code by conveniently and efficiently collecting runtime state during program execution.

After your MLE code has finished executing, debug data collected can be used to analyze program behavior and discover and fix bugs. This form of debugging is known as post-execution debugging.

The post-execution debug option allows you to instrument your code with debugpoints. Debugpoints allow for the logging of program state conditionally or unconditionally, including values of individual variables as well as execution snapshots. Debugpoints are specified as

JSON documents separate from the application code. No change to the application code is necessary for debugpoints to fire.

When activated, debug information is collected according to the debug specification and can be fetched for later analysis by a wide range of tools thanks to its standard format.

> **See Also:**
>
> Post-Execution Debugging of MLE JavaScript Modules for more details about post-execution debugging with MLE