Oracle® Database

JavaScript Developer's Guide

Release 23ai
F56885-13
May 2025

Oracle Database JavaScript Developer's Guide, Release 23ai

F56885-13

Primary Author: Sarah Hirschfeld

Contributors: M. Bach, L. Braun-Lohrer, H. Kasture, A. Ulrich, G. Venzl, M. Brantner, L. Daynes, H. Guiroux, A. Schubert, A. Burlison, M. Keppner, A. Kashuba, N. Sheikh, A.A. Baha, D. Adams

Contents

7 Calling PL/SQL and SQL from the MLE JavaScript SQL Driver

8 Working with SODA Collections in MLE JavaScript Code

9 Post-Execution Debugging of MLE JavaScript Modules

10 MLE Security

A MLE Type Conversions

Index

List of Examples

List of Figures

List of Tables

1

Changes in This Release for JavaScript Developer's Guide

This chapter lists the changes in *Oracle Database JavaScript Developer's Guide* for Oracle Database 23ai:

- July 2024, Release Update 23.5
- January 2025, Release Update 23.7
- April 2025, Release Update 23.8

July 2024, Release Update 23.5

Included are some notable *Oracle Database JavaScript Developer's Guide* updates with Oracle Database 23ai, Release Update 23.5:

Feature	Description
MLE Support on Linux for Arm (aarch64)	In addition to Linux x86-64, Multilingual Engine (MLE) is supported on Linux for Arm (aarch64). Overview of Multilingual Engine for JavaScript
Operator Overloading with `OracleNumber`	Rather than using methods such as `add` and `sub` to perform arithmetic operations with instances of the type `OracleNumber`, arithmetic operators such as `+` and `-` are now supported as well. Examples using this new syntax can be found in Type Mapping. Server-Side JavaScript API Documentation

January 2025, Release Update 23.7

Included are some notable *Oracle Database JavaScript Developer's Guide* updates with Oracle Database 23ai, Release Update 23.7:

Feature	Description
Foreign Function Interface	The Foreign Function Interface (FFI) allows you to handle PL/SQL packages, functions, and procedures as JavaScript objects, providing more direct access to objects created in PL/SQL. Introduction to the PL/SQL Foreign Function Interface Server-Side JavaScript API Documentation
Fetch Type Handler	The `fetchTypeHandler` property of `mle-js-oracledb` is available to modify query result sets in JavaScript. Using the fetch type handler you can, for example, change the data types of the resulting row(s) of a `SELECT` statement. Server-Side JavaScript API Documentation

April 2025, Release Update 23.8

Included are some notable *Oracle Database JavaScript Developer's Guide* updates with Oracle Database 23ai, Release Update 23.8:

Feature	Description
Restricted JavaScript Execution Contexts	The `PURE` keyword is used in the creation of MLE environments and in inline call specifications to specify the use of a restricted execution context. During PURE execution, JavaScript code cannot access database state. About Restricted Execution Contexts

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

3

MLE JavaScript Modules and Environments

A JavaScript module is a unit of MLE's language code stored in the database as a schema object.

Storing code within the database is one of the main benefits of using JavaScript in MLE: rather than having to manage a fleet of application servers each with their own copy of the application, the database takes care of this for you.

In addition, Data Guard replication ensures that the same code is present in both production and all physical standby databases. Configuration drift, a common problem bound to occur when invoking the disaster recovery location, can be mitigated against.

A JavaScript module in MLE is equivalent to an ECMAScript 6 module. The terms MLE module and JavaScript module are used interchangeably. The contents are specific to JavaScript and can be managed using Data Definition Language (DDL) commands.

In traditional JavaScript environments, additional information is often passed to the runtime using directives or configuration scripts. In MLE, this can be achieved using MLE environments, an additional metadata structure complementing MLE modules. MLE environments are also used for name resolution of JavaScript module imports. Name resolution is crucial for maintaining code and separating it into various modules to be used with MLE.

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

Topics

Using JavaScript Modules in MLE

JavaScript modules can be used in several different ways and can be managed using a set of Data Definition Language (DDL) commands.

JavaScript code provided in MLE modules can be used in the following ways:

- JavaScript functions exported by an MLE modules can be published by creating a call specification known as an MLE module call. This allows the function to be called directly from SQL and PL/SQL.

- Functionality exported by a JavaScript MLE module can be imported in other MLE JavaScript modules.
- Code snippets in `DBMS_MLE` can import modules for dynamic invocation of JavaScript.

Before a user can create and execute MLE modules, several privileges must be granted.

> **See Also:**
>
> - Overview of Importing MLE JavaScript Modules for more information about module calls
> - Overview of Dynamic MLE Execution for more information about `DBMS_MLE` and dynamic invocation of JavaScript code in the database
> - System and Object Privileges Required for Working with JavaScript in MLE for more information about MLE-specific privileges

Topics

- Managing JavaScript Modules in the Database
  SQL allows the creation of MLE modules as schema objects, assuming the necessary privileges are in place.
- Preparing JavaScript code for MLE Module Calls
  JavaScript modules in MLE follow the ECMAScript 6 standard for modules. Functions and variables expected to be consumed by users of the MLE module must be exported.
- Additional Options for Providing JavaScript Code to MLE
  The JavaScript source code of an MLE module can be specified inline with PL/SQL but can also be provided using a BFILE, BLOB, or CLOB, in which case the source file must be UTF8 encoded.
- Specifying Module Version Information and Providing JSON Metadata
  MLE modules may carry optional metadata in the form of a version string and free-form JSON-valued metadata.
- Drop JavaScript Modules
  The `DROP MLE MODULE` DDL statement is used for dropping an MLE module.
- Alter JavaScript Modules
  Attributes of an MLE module can be assigned or altered using the `ALTER MLE MODULE` statement.
- Overview of Built-in JavaScript Modules
  MLE provides a set of built-in JavaScript modules that are available for import in any execution context.
- Dictionary Views Related to MLE JavaScript Modules
  The Data Dictionary includes details about JavaScript modules.

Managing JavaScript Modules in the Database

SQL allows the creation of MLE modules as schema objects, assuming the necessary privileges are in place.

At a minimum, you need the `CREATE MLE MODULE` privilege to create or replace an MLE module in your own schema. Additionally, you must have the execute privilege on the target JavaScript language object.

> **See Also:**
>
> - System and Object Privileges Required for Working with JavaScript in MLE for more information about MLE-specific privileges
> - *Oracle Database Security Guide* for more details about privileges and roles in Oracle Database

Topics

- Naming JavaScript Modules
  Each JavaScript module name must be unique in the schema that it is created in. Unless a fully qualified name is used, the current user's schema is used.
- Creating JavaScript Modules in the Database
  JavaScript modules are created in the database using the `CREATE MLE MODULE` DDL statement, specifying name and source code of the MLE module.
- Storing JavaScript Code in Databases Using Single-Byte Character Sets
  Character set standards and things to remember when using a single-byte character set with MLE.
- Code Analysis
  JavaScript syntax errors are flagged when an MLE module is created but a linting tool of your choice should still be used to perform analysis before executing the `CREATE MLE MODULE` command.

Naming JavaScript Modules

Each JavaScript module name must be unique in the schema that it is created in. Unless a fully qualified name is used, the current user's schema is used.

As with other schema object identifiers, the module name is case-sensitive if enclosed in double quotation marks. If the enclosing quotation marks are omitted, the name is implicitly converted to uppercase.

When choosing a unique name, note that MLE objects share the namespace with tables, views, materialized views, sequences, private synonyms, PL/SQL packages, functions, procedures, and cache groups.

Creating JavaScript Modules in the Database

JavaScript modules are created in the database using the `CREATE MLE MODULE` DDL statement, specifying name and source code of the MLE module.

As soon as an MLE module has been created, it is persisted in the database dictionary. This is one of the differences when compared with dynamic execution of JavaScript code using `DBMS_MLE`.

`CREATE MLE MODULE` (without the `OR REPLACE` clause) throws an error if an MLE module with the given name already exists. With `CREATE OR REPLACE MLE MODULE`, the existing module is replaced if it exists, otherwise a new one is created. When an MLE module is replaced, privileges to it do not need to be re-granted.

For those who are familiar with PL/SQL, note that this is exactly the same behavior experienced with PL/SQL program units.

If you do not wish to replace an existing module in the event the module name is already in use, you can use the `IF NOT EXISTS` clause rather than `CREATE OR REPLACE`. The syntax for this variation is shown in Example 3-1. The `IF NOT EXISTS` and `OR REPLACE` clauses are mutually exclusive.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for the complete `CREATE MLE MODULE` syntax
> - *Oracle Database Development Guide* for more information about using the `IF [NOT] EXISTS` syntax

Example 3-1 Creating a JavaScript Module in the Database

This example demonstrates the creation of an MLE module and the export of a simple JavaScript function.

```
CREATE MLE MODULE IF NOT EXISTS po_module LANGUAGE JAVASCRIPT AS

/**
* get the value of all line items in an order
* @param {array} lineItems - all the line items in a purchase order
* @returns {number} the total value of all line items in a purchase order
*/
export function orderValue(lineItems) {

    return lineItems
        .map( x => x.Part.UnitPrice * x.Quantity )
        .reduce(
            (accumulator, currentValue) => accumulator + currentValue, 0
        );
}
/
```

The first line of this code block specifies the JavaScript module name as `po_module`. The remaining lines define the actual JavaScript code. Note that in line with the ECMAScript standard, the export keyword indicates the function to be exported to potential callers of the module. MLE accepts code adhering to the ECMAScript 2023 standard.

Storing JavaScript Code in Databases Using Single-Byte Character Sets

Character set standards and things to remember when using a single-byte character set with MLE.

JavaScript is encoded in Unicode. The Unicode Standard is a character encoding system that defines every character in most of the spoken languages in the world. It was developed to overcome limitations of other character-set encodings.

Oracle recommends creating databases using the AL32UTF8 character set. Using the AL32UTF8 character set in the database ensures the use of the latest version of the Unicode Standards and minimizes the potential for character-set conversion errors.

In case your database still uses a single-byte character set such as US7ASCII, WE8ISO8859-n, or WE8MSWIN1252, you must be careful not to use Unicode features in MLE JavaScript code. This is no different than handling other types of input data with such a database.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for more details about the Unicode Standard

Code Analysis

JavaScript syntax errors are flagged when an MLE module is created but a linting tool of your choice should still be used to perform analysis before executing the `CREATE MLE MODULE` command.

When creating MLE modules in the database, you should use a well-established toolchain in the same way other JavaScript projects are governed. In this sense, the call to `CREATE MLE MODULE` can be considered a deployment step, similar to deploying a server application. Code checking should be performed during a build step, for example by a continuous integration and continuous deployment (CI/CD) pipeline, prior to deployment.

If a module is created using `CREATE MLE MODULE` that includes syntax errors in the JavaScript code, the module will be created but it will exist in an invalid state. This check does not apply to any SQL statements called within the module, so separate testing should still be performed to ensure that the code works as expected.

It is considered an industry best practice to process code with a tool called a *linter* before checking it into a source-code repository. As with any other development project, you are free to choose the best option for yourself and your team. Some potential options include ESLint, JSHint, JSLint, and others that perform static code analysis to flag syntax errors, bugs, or otherwise problematic code. They can also be used to enforce a certain coding style. Many integrated development environments (IDEs) provide linting as a built-in feature, invoking the tool as soon as a file is saved to disk and flagging any issues.

In addition to executing linting dynamically, it is possible to automate the code analysis using highly automated DevOps environments to invoke linting as part of a build pipeline. This step usually occurs prior to submitting the JavaScript module to the database.

The aim is to trap as many potential issues as possible before they can produce problems at runtime. Unit tests can help further mitigate these risks and their inclusion into the development process have become an industry best practice. Regardless of the method you choose, the code analysis step occurs prior to submitting the JavaScript module to the database.

Preparing JavaScript code for MLE Module Calls

JavaScript modules in MLE follow the ECMAScript 6 standard for modules. Functions and variables expected to be consumed by users of the MLE module must be exported.

Those variables and functions *not* exported are considered private in the module. Example 3-3 demonstrates the use of both public and private functions in an MLE JavaScript module.

An ECMAScript module can import other ECMAScript modules using import statements or dynamic import calls. This functionality is present in MLE as well. Complementary metadata to MLE modules is provided in MLE environments.

Note that console output in MLE is facilitated using the console object. By default, anything written to `console.log()` is routed to `DBMS_OUTPUT` and will end up on the screen.

JavaScript code like that in Example 3-1 cannot be accessed from SQL or PL/SQL without the help of call specifications. For now, you can think of a call specification as a PL/SQL program unit (function, procedure, or package) where its PL/SQL body is replaced with a reference to the JavaScript module and function, as shown in Example 3-2. For more information about call specifications, see MLE JavaScript Functions.

> **See Also:**
>
> Using MLE Environments for Import Resolution

Example 3-2 Create a Call Specification for a Public Function

This example uses the module `po_module` created in Example 3-1. A call specification for `orderValue()`, the only function exported in `po_module`, can be written as follows:

```
CREATE OR REPLACE FUNCTION order_value(
    p_line_items JSON
) RETURN NUMBER AS
MLE MODULE po_module
SIGNATURE 'orderValue';
/
```

Once the function is created, it is possible to calculate the value of a given purchase order:

```
SELECT
    po.po_document.PONumber,
    order_value(po.po_document.LineItems[*]) order_value
```

```
FROM
    j_purchaseorder po;
```

Result:

```
PONUMBER   ORDER_VALUE
---------- -----------
1600             279.3
672              359.5
```

Example 3-3 Public and Private Functions in a JavaScript Module

In addition to public (exported) functions, it is possible to add functions private to the module. In this example, the calculation of the value is taken out of the `map()` function and moved to a separate function (refactoring).

The first function in the following code, `lineItemValue()`, is considered private, whereas the second function, `orderValue()`, is public. The export keyword is provided at the end of this code listing but can also appear as a prefix for variables and functions, as seen in Example 3-1. Both variations are valid JavaScript syntax.

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

/**
* calculate the value of a given line item. Factored out of the public
* function to allow for currency conversions in a later step
* @param {number} unitPrice - the price of a single article
* @param {number} quantity - the quantity of articles ordered
* @returns {number} the monetary value of the line item
*/
function lineItemValue(unitPrice, quantity) {
    return unitPrice * quantity;
}

/**
* get the value of all line items in an order
* @param {array} lineItems - all the line items in a purchase order
* @returns {number} the total value of all line items in a purchase order
*/
function orderValue(lineItems) {

    return lineItems
                .map( x => lineItemValue(x.Part.UnitPrice, x.Quantity) )
                .reduce(
                    (accumulator, currentValue) => accumulator +
currentValue, 0
                );
}

export { orderValue }
/
```

Additional Options for Providing JavaScript Code to MLE

The JavaScript source code of an MLE module can be specified inline with PL/SQL but can also be provided using a BFILE, BLOB, or CLOB, in which case the source file must be UTF8 encoded.

Creating MLE modules using the BFILE clause can cause problems with logical replication such as GoldenGate. In order for the DDL command to succeed on the target database, the same directory must exist on the target database. Furthermore, the same JavaScript file must be present in this directory. Failure to adhere to these conditions will cause the call to create the MLE module on the target database to fail.

A BLOB or a CLOB can also be used to create an MLE module as an alternative to using a BFILE. Example 3-5 shows how to create a JavaScript module using a CLOB. If you prefer to use a BLOB, the syntax is the same but the value of the BLOB will differ from that of a CLOB.

Another option available is to use the `mle create-module` command with SQLcl to load a JavaScript file into the database. Because the module is being created directly from a JavaScript file, there is no need to sandwich the JavaScript code between DDL statements. This means regular programming steps such as linting, local unit testing, and the use of formatting tools can be performed more conveniently. The use of SQLcl can be particularly well suited for Continuous Integration (CI) pipelines.

Example 3-4 Providing JavaScript Source Code Using a BFILE

In this example, `JS_SRC_DIR` is a database directory object mapping to a location on the local file system containing the module's source code in a file called `myJavaScriptModule.js`. When loading the file from the directory location, MLE stores the source code in the dictionary. Subsequent calls to the MLE module will not cause the source code to be refreshed from the disk. If there is a new version of the module stored in `myJavaScriptModule.js`, it must be deployed using another call to `CREATE OR REPLACE MLE MODULE`.

```
CREATE MLE MODULE mod_from_bfile
LANGUAGE JAVASCRIPT
USING BFILE(JS_SRC_DIR,'myJavaScriptModule.js');
/
```

Example 3-5 Providing JavaScript Source Code Using a CLOB

```
CREATE OR REPLACE MLE MODULE mod_from_clob_inline
LANGUAGE JAVASCRIPT USING CLOB (
    SELECT q'~
    export function clob_hello(who){
        return `hello, ${who}`;
}
~')
/
```

As an alternative, you also have the option of using JavaScript source code that is stored in a table. This example variation assumes your schema features a table named `javascript_src`

containing the JavaScript source code in column `src` along with some additional metadata. The following statement fetches the CLOB and creates the module.

```
CREATE OR REPLACE MLE MODULE mod_from_clob_table
LANGUAGE JAVASCRIPT USING CLOB (
    SELECT src
    FROM javascript_src
    WHERE
        id = 1 AND
        commit_hash = 'ac1fd40'
)
/
```

Staging tables like this can be found in environments where Continuous Integration (CI) pipelines are used to deploy JavaScript code to the database.

Example 3-6 Providing JavaScript Source Code Using SQLcl

In this example, the module's source code is in a file called `myJavaScriptModule.js`, which is located in a local file directory folder called `tmp`. The following command creates a module called `my_js_mod`, replacing the module if it exists or creating one if it does not.

```
mle create-module -
  -language javascript -
  -replace -
  -filename /tmp/myJavaScriptModule.js -
  -module-name my_js_mod
```

For more information about SQLcl MLE commands and their syntax, see *Oracle SQLcl User's Guide*.

Specifying Module Version Information and Providing JSON Metadata

MLE modules may carry optional metadata in the form of a version string and free-form JSON-valued metadata.

Both kinds of metadata are purely informational and do not influence the behavior of MLE. They are stored alongside the module in the data dictionary.

The `VERSION` flag can be used as an internal reminder about what version of the code is deployed. The information stored in the `VERSION` field allows developers and administrators to identify the code in the version control system.

The format for JSON metadata is not bound to a schema; anything useful or informative can be added by the developer. In case the MLE module is an aggregation of sources created by a tool such as `rollup.js` or webpack, it can be useful to store the associated `package-lock.json` file alongside the module.

The metadata field can be used to create a software bill of material (SBOM), allowing security teams and administrators to track information about deployed packages, especially in the case where third-party modules are used.

Tracking dependencies and vulnerabilities in the upstream repository supports easier identification of components in need of update after security vulnerabilities have been reported.

> **See Also:**
>
> - Dictionary Views Related to MLE JavaScript Modules
> - Software Bill of Material for more information about using the metadata field to store a SBOM

Example 3-7 Specification of a VERSION string in CREATE MLE MODULE

```
CREATE OR REPLACE MLE MODULE version_mod
 LANGUAGE JAVASCRIPT
 VERSION '1.0.0.1.0'
AS
export function sq(num) {
  return num * num;
}
/
```

Example 3-8 Addition of JSON Metadata to the MLE Module

This example uses the module `version_mod`, created in Example 3-7.

```
ALTER MLE MODULE version_mod
SET METADATA USING CLOB
(SELECT
  '{
    "name": "devel",
    "lockfileVersion": 2,
    "requires": true,
    "packages": {}
  }'
)
/
```

Drop JavaScript Modules

The `DROP MLE MODULE` DDL statement is used for dropping an MLE module.

The `DROP` statement specifies the name, and optionally the schema of the module to be dropped. If a schema is not specified, the schema of the current user is assumed.

Attempting to drop an MLE module that does not exist causes an error to be thrown. In cases where this is not desirable, the `IF EXISTS` clause can be used. The `DROP MLE MODULE` command is silently skipped if the indicated MLE module does not exist.

Example 3-9 Drop an MLE Module

```
DROP MLE MODULE unused_mod;
```

Example 3-10 Drop an MLE Module Using IF EXISTS

```
DROP MLE MODULE IF EXISTS unused_mod;
```

Alter JavaScript Modules

Attributes of an MLE module can be assigned or altered using the `ALTER MLE MODULE` statement.

The `ALTER MLE MODULE` statement specifies the name, and optionally the schema of the module to be altered. If the module name is not prefixed with a schema, the schema of the current user is assumed.

Example 3-11 Alter an MLE Module

```
ALTER MLE MODULE change_mod
  SET METADATA USING CLOB(SELECT'{...}');
```

Overview of Built-in JavaScript Modules

MLE provides a set of built-in JavaScript modules that are available for import in any execution context.

Built-in modules are not deployed to the database as user-defined MLE modules, but are included as part of the MLE runtime. In particular, MLE provides the following three built-in JavaScript modules:

- **`mle-js-oracledb`** is the JavaScript MLE SQL Driver.
- **`mle-js-bindings`** provides functionality to import and export values from the PL/SQL engine.
- **`mle-js-plsqltypes`** provides definitions for the PL/SQL wrapper types. For example, JavaScript types that wrap PL/SQL and SQL types like `OracleNumber`.
- **`mle-js-fetch`** provides a partial Fetch API polyfill, allowing developers to invoke external resources.
- **`mle-encode-base64`** contains code to work with base64-encoded data.
- **`mle-js-encodings`** provides functionality to handle text in UTF-8 and UTF-16 encodings.
- **`mle-js-plsql-ffi`** provides functionality to handle PL/SQL packages, functions, and procedures as JavaScript objects.

These modules can be used to interact with the database and provide type conversions between the JavaScript engine and database engine.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the built-in JavaScript modules

Dictionary Views Related to MLE JavaScript Modules

The Data Dictionary includes details about JavaScript modules.

Topics

- USER_SOURCE
  Each JavaScript module's source code is externalized using the `[USER | ALL | DBA | CDB]_SOURCE` dictionary views.
- USER_MLE_MODULES
  Metadata pertaining to JavaScript MLE modules are found in `[USER | ALL | DBA | CDB]_MLE_MODULES`.

USER_SOURCE

Each JavaScript module's source code is externalized using the `[USER | ALL | DBA | CDB]_SOURCE` dictionary views.

Modules created with references to the file system using the BFILE operator show the code at the time of the module's creation.

For more information about `*_SOURCE`, see *Oracle Database Reference*.

Example 3-12 Externalize JavaScript Module Source Code

```
SELECT
    line,
    text
FROM
    USER_SOURCE
WHERE
    name = 'PO_MODULE';
```

Example output:

```
 LINE TEXT
----- ------------------------------------------------------------
    1 /**
    2  * calculate the value of a given line item. Factored out of the public
    3  * function to allow for currency conversions in a later step
    4  * @param {number} unitPrice - the price of a single article
    5  * @param {number} quantity - the quantity of articles ordered
    6  * @returns {number} the monetary value of the line item
    7  */
    8 function lineItemValue(unitPrice, quantity) {
    9     return unitPrice * quantity;
   10 }
   11
   12
   13 /**
   14  * get the value of all line items in an order
   15  * @param {array} lineItems - all the line items in a purchase order
   16  * @returns {number} the total value of all line items in a purchase
```

```
order
   17  */
   18 export function orderValue(lineItems) {
   19
   20     return lineItems
   21                 .map( x => lineItemValue(x.Part.UnitPrice, x.Quantity) )
   22                 .reduce(
   23                     (accumulator, currentValue) => accumulator +
currentValue, 0
   24                 );
   25 }
```

USER_MLE_MODULES

Metadata pertaining to JavaScript MLE modules are found in `[USER | ALL | DBA | CDB]_MLE_MODULES`.

Any JSON metadata specified, version information, as well as language, name, and owner can be found in this view.

For more information about `*_MLE_MODULES`, see *Oracle Database Reference*.

Example 3-13 Find MLE Modules Defined in a Schema

```
SELECT MODULE_NAME, VERSION, METADATA
FROM USER_MLE_MODULES
WHERE LANGUAGE_NAME='JAVASCRIPT'
/
```

Example output:

```
MODULE_NAME                    VERSION    METADATA
------------------------------ ---------- -----------
MY_MOD01                       1.0.0.1
MY_MOD02                       1.0.1.1
MY_MOD03
```

Specifying Environments for MLE Modules

MLE environments are schema objects in the database. Their functionality and management methods are described.

MLE environments complement MLE modules and allow you to do the following:

- Set language options to customize the JavaScript runtime in its execution context
- Enable specific MLE modules to be imported
- Manage name resolution and the import chain

Topics

- Creating MLE Environments in the Database
  The SQL DDL supports the creation of MLE environments.

- Dropping MLE Environments
  MLE environments that are no longer needed can be dropped using the `DROP MLE ENV` command.
- Modifying MLE Environments
  Existing MLE environments can be modified using the `ALTER MLE ENV` command.
- Dictionary Views Related to MLE JavaScript Environments
  Details about MLE environments are available in these families of views: `USER_MLE_ENVS` and `USER_MLE_ENV_IMPORTS`.

Creating MLE Environments in the Database

The SQL DDL supports the creation of MLE environments.

Just like MLE modules, MLE environments are schema objects in the database, persisted in the data dictionary.

At a minimum, you must have the `CREATE MLE MODULE` privilege to create or replace an MLE environment in your own schema.

> **See Also:**
>
> - System and Object Privileges Required for Working with JavaScript in MLE for more information about the privileges necessary to create and execute JavaScript code in MLE
> - *Oracle Database Security Guide* for details about privileges and roles in Oracle Database

Topics

- Naming MLE Environments
  Each JavaScript environment's name must be unique in the schema it is created in. Unless a fully qualified name is used, the current user's schema is used.
- Creating an Empty MLE Environment
  The DDL statement `CREATE MLE ENV` can be used to create an MLE environment.
- Creating an Environment as a Clone of an Existing Environment
  If needed, a new environment can be created as a point-in-time copy of an existing environment.
- Using MLE Environments for Import Resolution
  It is possible to import functionality exported by one JavaScript module into another using the import statement.
- Providing Language Options
  MLE allows the customization of JavaScript's runtime by setting language-specific options in MLE environments.

Naming MLE Environments

Each JavaScript environment's name must be unique in the schema it is created in. Unless a fully qualified name is used, the current user's schema is used.

As with other schema object identifiers, the name is case-sensitive if enclosed in double quotation marks. If the enclosing quotation marks are omitted, the name is implicitly converted to uppercase.

MLE environments cannot contain import mappings that conflict with the names of the MLE built-in modules (`mle-js-oracledb`, `mle-js-bindings`, `mle-js-plsqltypes`, `mle-js-fetch`, `mle-encode-base64`, `mle-js-encodings`, and `mle-js-plsql-ffi`). If you attempt to add such a mapping using either the `CREATE MLE ENV` or `ALTER MLE ENV` DDL, the operation fails with an error.

Creating an Empty MLE Environment

The DDL statement `CREATE MLE ENV` can be used to create an MLE environment.

In its most basic form, an environment can be created empty as shown in the following snippet:

```
CREATE MLE ENV myEnv;
```

Subsequent calls to `ALTER MLE ENV` can be used to add properties to the environment.

Just like with MLE modules, it is possible to append the `OR REPLACE` clause to instruct the database to replace an existing MLE environment rather than throwing an error.

Furthermore, the `IF NOT EXISTS` clause can be used instead of the `OR REPLACE` clause to prevent the creation of a new MLE environment in the case that one already exists with the same name. In this case, the statement used to create the environment changes to the following:

```
CREATE MLE ENV IF NOT EXISTS myEnv;
```

> **Note:**
>
> The `IF NOT EXISTS` and `OR REPLACE` clauses are mutually exclusive.

You can optionally include the `PURE` keyword to indicate that any JavaScript code using the environment should be run in a restricted execution context that disallows access to the database state. PURE execution provides an extra layer of security by isolating certain code, such as third-party JavaScript libraries, from the database. Environments that are created using the `PURE` keyword can be referenced by MLE modules and when using `DBMS_MLE` for dynamic execution. The `PURE` keyword can be specified as follows:

```
CREATE OR REPLACE MLE ENV my_pure_env PURE;
```

> **See Also:**
>
> Modifying MLE Environments for information about editing existing environments
>
> About Restricted Execution Contexts for information about the `PURE` keyword and restricted contexts
>
> *Oracle Database SQL Language Reference* for the full syntax of `CREATE MLE ENV`

Creating an Environment as a Clone of an Existing Environment

If needed, a new environment can be created as a point-in-time copy of an existing environment.

The new environment inherits all settings from its source. Subsequent changes to the source are not propagated to the clone. A clone can be created as shown in the following statement:

```
CREATE MLE ENV MyEnvDuplicate CLONE MyEnv
```

Using MLE Environments for Import Resolution

It is possible to import functionality exported by one JavaScript module into another using the import statement.

The separation of code allows for finer control over changes and the ability to write more reusable code. Simplified code maintenance is another positive effect of this approach.

Only those identifiers marked with the export keyword are eligible for importing.

Modules attempting to import functionality from other modules stored in the database require MLE environments in order to perform name resolution. To create an MLE environment with that information, the `IMPORTS` clause must be used. Example 3-14 demonstrates how a mapping is created between the identifier `po_module` and JavaScript module `PO_MODULE`, created in Example 3-1.

Multiple imports can be provided as a comma-separated list. In Example 3-14, the first parameter in single quotation marks is known as the import name. The import name is used by another module's import statement. In this case, `'po_module'` is the import name and refers to the module of the same name.

> **Note:**
>
> The import name does not have to match the module name. Any valid JavaScript identifier can be used. The closer the import name matches the module name it refers to, the easier it is to identify the link between the two.

The `CREATE MLE ENV` command fails if a module referenced in the `IMPORTS` clause does not exist or is not accessible to you.

Built-in JavaScript modules can be imported directly without having to specify additional MLE environments.

> **See Also:**
>
> Overview of Built-in JavaScript Modules for more information about built-in modules

Example 3-14 Map Identifier to JavaScript Module

```
CREATE OR REPLACE MLE ENV
    po_env
IMPORTS (
    'po_module' MODULE PO_MODULE
);
```

Example 3-15 Import Module Functionality

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

import * as po from "po_module";
/**
* use po_module's getValue() function to calculate the value of
* a purchase order. In later chapters, when discussing the MLE
* JavaScript SQL driver the hard-coded value used as the PO will
* be replaced by calls to the database
* @returns {number} the value of all line items in the purchase order
*/
export function purchaseOrderValue() {

    const purchaseOrder = {
        "PONumber": 1600,
        "Reference": "ABULL-20140421",
        "Requestor": "Alexis Bull",
        "User": "ABULL",
        "CostCenter": "A50",
        "ShippingInstructions": {
            "name": "Alexis Bull",
            "Address": {
                "street": "200 Sporting Green",
                "city": "South San Francisco",
                "state": "CA",
                "zipCode": 99236,
                "country": "United States of America"
            },
            "Phone": [
                {
                    "type": "Office",
                    "number": "909-555-7307"
                },
                {
                    "type": "Mobile",
                    "number": "415-555-1234"
                }
```

```
            ]
        },
        "Special Instructions": null,
        "AllowPartialShipment": true,
        "LineItems": [
            {
                "ItemNumber": 1,
                "Part": {
                    "Description": "One Magic Christmas",
                    "UnitPrice": 19.95,
                    "UPCCode": 13131092899
                },
                "Quantity": 9.0
            },
            {
                "ItemNumber": 2,
                "Part": {
                    "Description": "Lethal Weapon",
                    "UnitPrice": 19.95,
                    "UPCCode": 85391628927
                },
                "Quantity": 5.0
            }
        ]
    };

    return po.orderValue(purchaseOrder.LineItems);
}
/

purchaseOrderValue


CREATE FUNCTION purchase_order_value
RETURN NUMBER AS
MLE MODULE import_example_module
ENV po_env
SIGNATURE 'purchaseOrderValue';
/

SELECT purchase_order_value;
/
```

Result:

```
PURCHASE_ORDER_VALUE
--------------------
               279.3
```

Providing Language Options

MLE allows the customization of JavaScript's runtime by setting language-specific options in MLE environments.

Any options specified in the MLE environment take precedence over the default settings.

Multiple language options can be provided as a comma-separated list of `'<key>=<value>'` strings. The following snippet demonstrates how to enforce JavaScript's strict mode.

```
CREATE MLE ENV MyEnvOpt
    LANGUAGE OPTIONS 'js.strict=true';
```

Changes made to the language options of an environment are not propagated to execution contexts that have already been created using the environment. For changes to take effect for existing contexts, the contexts need to be dropped and recreated.

> **Note:**
>
> White space characters are not allowed between the key, equal sign, and value.

Topics

- JavaScript Language Options
  A full list of JavaScript language options available to be used with MLE are included.

JavaScript Language Options

A full list of JavaScript language options available to be used with MLE are included.

Table 3-1 JavaScript Language Options

Language Option	Accepted Value Type	Default	Description
`js.strict`	boolean	`false`	Enforce strict mode.
`js.console`	boolean	`true`	Provide `console` global property.
`js.polyglot-builtin`	boolean	`true`	Provide `Polyglot` global property.

Dropping MLE Environments

MLE environments that are no longer needed can be dropped using the `DROP MLE ENV` command.

The following snippet demonstrates a basic example of dropping an MLE module:

```
DROP MLE ENV myOldEnv;
```

As with MLE modules, the `IF EXISTS` clause prevents an error if the named MLE environment does not exist, as shown in the following snippet:

```
DROP MLE ENV IF EXISTS myOldEnv;
```

Modifying MLE Environments

Existing MLE environments can be modified using the `ALTER MLE ENV` command.

It is possible to modify language options and the imports clause.

Topics

- Altering Language Options
  You can modify language options provided to an MLE module.
- Modifying Module Imports
  In the context of MLE module imports, the `ALTER MLE ENV` command allows you to add additional imports as well as modify and drop existing imports.

Altering Language Options

You can modify language options provided to an MLE module.

Use the `ALTER MLE ENV` clause to modify language options, as shown in the following snippet:

```
ALTER MLE ENV MyEnvOpt
    SET LANGUAGE OPTIONS 'js.strict=false';
```

Modifying Module Imports

In the context of MLE module imports, the `ALTER MLE ENV` command allows you to add additional imports as well as modify and drop existing imports.

Imports not specified during an environment's creation can be added to existing MLE environments using the `ADD IMPORTS` clause. Import names, once defined, are static and must be dropped before they can be added as desired. Assuming you have run a new `CREATE MLE` DDL to replace `IMPORT_EXAMPLE_MODULE` from Example 3-1 with the module name `IMPORT_EXAMPLE_MODULE_V2`, the following statement will run successfully:

```
ALTER MLE ENV po_env
ADD IMPORTS (
    'import_example' MODULE IMPORT_EXAMPLE_MODULE_V2
);
```

Any imports no longer needed can be dropped using the `DROP IMPORTS` clause:

```
ALTER MLE ENV po_env DROP IMPORTS('import_example');
```

The case of the import identifier must match that in the data dictionary's `USER_MLE_ENV_IMPORTS` view.

Dictionary Views Related to MLE JavaScript Environments

Details about MLE environments are available in these families of views: `USER_MLE_ENVS` and `USER_MLE_ENV_IMPORTS`.

In addition to the `USER` prefix, these views exist in all namespaces: `CDB`, `DBA`, `ALL`, and `USER`.

Topics

- USER_MLE_ENVS
  The `USER_MLE_ENVS` view lists all MLE environments available to you along with the defined language options.
- USER_MLE_ENV_IMPORTS
  The `[USER | ALL | DBA | CDB]_MLE_ENV_IMPORTS` family of views lists imported modules.

USER_MLE_ENVS

The `USER_MLE_ENVS` view lists all MLE environments available to you along with the defined language options.

For more information about `*_MLE_ENVS`, see *Oracle Database Reference*.

Example 3-16 List Available MLE Environments Using USER_MLE_ENVS

```
SELECT ENV_NAME, LANGUAGE_OPTIONS
FROM USER_MLE_ENVS
WHERE ENV_NAME='MYENVOPT'
/
```

Example SQL*Plus output:

```
ENV_OWNER            ENV_NAME   LANGUAGE_OPTIONS
-------------------- ---------- ----------------------
JSDEV01              MYENVOPT   js.strict=true
```

USER_MLE_ENV_IMPORTS

The `[USER | ALL | DBA | CDB]_MLE_ENV_IMPORTS` family of views lists imported modules.

MLE environments are the key enablers for resolving names of imported modules. Example 3-17 demonstrates a query against `USER_MLE_ENV_IMPORTS` to list `IMPORT_NAME`, `MODULE_OWNER`, and `MODULE_NAME`.

For more information about `*_MLE_ENV_IMPORTS`, see *Oracle Database Reference*

Example 3-17 List Module Import Information Using USER_MLE_ENV_IMPORTS

```
SELECT IMPORT_NAME, MODULE_OWNER, MODULE_NAME
    FROM USER_MLE_ENV_IMPORTS
    WHERE ENV_NAME='MYFACTORIALENV';
/
```

SQL*Plus output:

```
IMPORT_NAME          MODULE_OWNER             MODULE_NAME
-------------------- ------------------------ ----------------
FACTORIAL_MOD        DEVELOPER1               FACTORIAL_MOD
```

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
```

5

Overview of Importing MLE JavaScript Modules

One of the key benefits of MLE is the ability to store modules of JavaScript code in the Oracle Database. The availability of modules supports self-contained and reusable code, key to developing successful software projects.

MLE modules interact with each other through imports and exports. Specifically, if a module wants to use a functionality provided by another module, the source module must be exported and then imported into the calling module's scope.

Due to a difference in architecture, module imports behave slightly differently in the Oracle Database when compared to other development environments. For example, JavaScript source code used with Node.js is stored in a specific directory structure on disk. Alternatively, MLE modules are stored together with the database, rather than in a file system, so must be referenced in a different manner.

There are two options available for module imports in MLE:

- Importing module functionality into another module
- Importing module functionality into a code snippet to be executed via dynamic MLE (using the `DBMS_MLE` PL/SQL package)

> **Note:**
>
> MLE supports a pure JavaScript implementation. Module exports and imports are facilitated as ECMAScript modules using the `export` and `import` keywords. Other JavaScript modularization technologies such as CommonJS and Asynchronous Module Definition (AMD) are not available.

> **See Also:**
>
> MLE JavaScript Modules and Environments for more information on MLE modules

Topics

- JavaScript Module Hierarchies
  The use of import names as opposed to file-system location to resolve to MLE modules is described.
- Export Functionality
  Commonly exported identifiers in native JavaScript modules include variables, constants, functions, and classes.
- Import Functionality
  The `import` keyword allows developers to import functionality that has been exported by a source module.

JavaScript Module Hierarchies

The use of import names as opposed to file-system location to resolve to MLE modules is described.

A typical Node.js or browser-based workflow requires a module import to be followed by its location in the file system. For example, the following line is a valid module import statement in Node.js:

```
import * as myMath from './myMath.mjs'
```

Used with Node.js, this statement would import `myMath`'s contents into the current scope.

However, because MLE JavaScript modules are stored in the database, there are no file-system paths to be used for identification. Rather, MLE uses import names instead that resolve to the desired module.

> **Note:**
>
> As soon as a module import is detected by the JavaScript runtime engine, `strict mode` is enforced.

Topics

- Resolving Import Names Using MLE Environments
  Rather than file-system locations, MLE uses so-called import names instead. Import names must be valid JavaScript identifiers, but otherwise can be chosen freely.

Resolving Import Names Using MLE Environments

Rather than file-system locations, MLE uses so-called import names instead. Import names must be valid JavaScript identifiers, but otherwise can be chosen freely.

Example 5-1 Use an MLE Environment to Map an Import Name to a Module

This example shows how you might use an import name for code referencing functionality in module `named_exports_module`, which is defined in Example 5-2.

MLE in Oracle Database requires a link between the import name, `namedExports`, and the corresponding MLE module, `named_exports_module`, at runtime. Just as with import names, you can choose any valid name for the MLE environment. A potential mapping is shown in the following snippet. See Example 5-6 for a complete definition of module `mod_object_import_mod`.

```
CREATE OR REPLACE MLE ENV named_exports_env
    imports ('namedExports' MODULE named_exports_module);
/

CREATE OR REPLACE MLE MODULE mod_object_import_mod LANGUAGE JAVASCRIPT AS

import * as myMath from "namedExports";
```

```
function mySum() {...}
/
```

Export Functionality

Commonly exported identifiers in native JavaScript modules include variables, constants, functions, and classes.

Topics

- Named Exports
  The explicit use of identifiers in an export statement is referred to as using named exports in JavaScript.
- Default Exports
  As an alternative to named exports, a default export can be defined in JavaScript. A default export differs syntactically from a named export. Contrary to the latter, a default export does not require a set of curly brackets.
- Private Identifiers
  Any identifier not exported from a module is considered private and cannot be referenced outside the module's scope or in module call specifications.

Named Exports

The explicit use of identifiers in an export statement is referred to as using named exports in JavaScript.

Example 5-2 demonstrates the export of multiple functions using named exports.

Example 5-2 Function Export using Named Exports

This code snippet creates a module called `named_exports_module`, defines two functions, `sum()` and `difference()`, and then uses a named export to provide access for other modules to import the listed functions.

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

function sum(a, b) {
    return a + b;
}

function difference(a, b) {
    return a - b;
}

export {sum, difference};
/
```

Make note of the `export{}` statement at the end of the module. Named exports require the use of curly brackets when listing identifiers. Any identifier placed between the curly brackets is exported. Those not listed are not exported.

Rather than using the `export` statement towards the end of the module, it is also possible to prefix an identifier with the `export` keyword inline. The following example shows how the same

module from the previous example can be rewritten with the `export` keyword provided inline with the JavaScript code.

Example 5-3 Function Export Using Export Keyword Inline

This code snippet creates a module called `inline_export_module` and defines two functions, `sum()` and `difference()`, which are both prefaced with the `export` keyword inline.

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

export function sum(a, b) {
    return a + b;
}

export function difference(a, b) {
    return a - b;
}
/
```

Both `named_exports_module` from Example 5-2 and `inline_export_module` are semantically identical. The method used to export the functions is the only syntactical difference.

Default Exports

As an alternative to named exports, a default export can be defined in JavaScript. A default export differs syntactically from a named export. Contrary to the latter, a default export does not require a set of curly brackets.

> **Note:**
>
> In line with the ECMAScript standard, only one default export is possible per module.

Example 5-4 Export a Class Using a Default Export

The following code snippet creates a module called `default_export_module`, defines a class called `myMath`, and defaults the class using a default export.

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

export default class myMath {

    static sum(operand1, operand2) {
        return operand1 + operand2;
    }

    static difference(operand1, operand2) {
        return operand1 - operand2;
    }
}
/
```

Private Identifiers

Any identifier not exported from a module is considered private and cannot be referenced outside the module's scope or in module call specifications.

Example 5-5 Named Export of Single Function

The following code snippet creates a module called `private_export_module`, defines two functions, `sum()` and `difference()`, and exports the function `sum()` via named export. The function `difference()` is not included in the export statement, thus is only available within its own module's scope.

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

function sum(a, b) {
    return a + b;
}

function difference(a, b) {
    return a - b;
}

export { sum };
/
```

Import Functionality

The `import` keyword allows developers to import functionality that has been exported by a source module.

Topics

- Module Objects
  A module object supplies a convenient way to import everything that has been exported by a module.
- Named Imports
  The ECMAScript standard specifies named imports. Rather than using an import name, you also have the option to specify identifiers.
- Default Imports
  Unlike named imports, default imports don't require the use of curly braces. This syntactical difference is also relevant to MLE's built-in modules.

Module Objects

A module object supplies a convenient way to import everything that has been exported by a module.

The module object provides a means to access all identifiers exported by a module and is used as a kind of namespace when referring to the imports.

Example 5-6 Module Object Definition

```
CREATE MLE ENV named_exports_env
    IMPORTS('namedExports' module named_exports_module);

CREATE OR REPLACE MLE MODULE mod_object_import_mod
LANGUAGE JAVASCRIPT AS

//the definition of a module object is demonstrated by the next line
import * as myMath from "namedExports"

function mySum(){
    const result = myMath.sum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}

function myDifference(){
    const result = myMath.difference(4, 2);
    console.log(`the difference between 4 and 2 is ${result}`);
}

export {mySum, myDifference};
/
```

`myMath` identifies the module object and `named_exports_module` is the module name. Both `sum()` and `difference()` are available under the `myMath` scope in `mod_object_import_mod`.

Named Imports

The ECMAScript standard specifies named imports. Rather than using an import name, you also have the option to specify identifiers.

Example 5-7 Named Imports Using Specified Identifiers

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

import {sum, difference} from "namedExports";

function mySum(){
    const result = sum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}

function myDifference(){
    const result = difference(4, 2);
    console.log(`the difference between 4 and 2 is ${result}`);
}

export {mySum, myDifference};
/
```

Namespace clashes can ensue if multiple modules export the same identifier. To avoid this issue, you can provide an alias in the `import` statement.

Example 5-8 Named Imports with Aliases

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

//note the use of aliases in the next line
import {sum as theSum, difference as theDifference} from "namedExports";

function mySum(){
    const result = theSum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}

function myDifference(){
    const result = theDifference(4, 2);
    console.log(`the difference between 4 and 2 is ${result}`);
}

export {mySum, myDifference};
/
```

Instead of referencing the exported functions by their original names, the alias is used instead.

Default Imports

Unlike named imports, default imports don't require the use of curly braces. This syntactical difference is also relevant to MLE's built-in modules.

Example 5-9 Default Import

This example demonstrates the default import of `myMathClass`.

```
CREATE OR REPLACE MLE ENV default_export_env
    IMPORTS('defaultExportModule' MODULE default_export_module);

CREATE MLE MODULE default_import_module LANGUAGE JAVASCRIPT AS

//note the lack of curly braces in the next line
import myMathClass from "defaultExportModule";

export function mySum(){
    const result = myMathClass.sum(4, 2);
    console.log(`the sum of 4 and 2 is ${result}`);
}
/
```

The same technique applies to the use of MLE's built-in modules such as the SQL driver. Example 5-10 demonstrates the use of the SQL driver in JavaScript code.

Example 5-10 Default Import with Built-in Module

```
CREATE MLE MODULE default_import_built_in_mod
LANGUAGE JAVASCRIPT AS

//note that there is no need to use MLE environments with built-in modules
```

```
import oracledb from "mle-js-oracledb";

export function hello(){
    const options = {
        resultSet: false,
        outFormat: oracledb.OUT_FORMAT_OBJECT
    };
    const bindvars = [];

const conn = oracledb.defaultConnection();
const result = conn.execute('select user', bindvars, options);
console.log(`hello, ${result.rows[0].USER}`);
}
/
```

Unlike other examples using custom JavaScript modules, no MLE environment is required for importing a built-in module.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the built-in JavaScript modules

6
MLE JavaScript Functions

This chapter introduces the use of call specifications to publish JavaScript functions so that they can be called from SQL and PL/SQL. MLE's support of `OUT` and `IN OUT` parameters is also discussed.

Topics

Call Specifications for Functions

Call specifications for modules and inline MLE call specifications allow you to implement JavaScript functionality.

Functions exported by an MLE JavaScript module can be published by creating call specifications. A JavaScript function published with a call specification can be called from anywhere a PL/SQL function or procedure can be called.

Alternatively, inline MLE call specifications can be used to embed JavaScript code directly in the DDL. This option can be advantageous when you want to quickly implement a simple functionality using JavaScript.

Topics

Creating a Call Specification for an MLE Module

MLE call specification creation uses the generic `CREATE FUNCTION RETURNS AS` or `CREATE PROCEDURE AS` syntax, followed by MLE specific syntax.

Example 6-1 Creating MLE Call Specifications

This example walks you through the creation of an MLE module that exports two functions, and the creation of call specifications to publish those functions.

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

    export function greet(str){
        console.log(`Hello, ${str}`);
    }
    export function concat(str1, str2){
        return str1 + str2;
    }
/
```

The MLE module `jsmodule` exports two functions. The function `greet()` takes an input string argument and prints a simple greeting, while the function `concat()` takes two strings as input and returns the concatenated string as the result.

Because `greet()` does not return a value, you must create a PL/SQL procedure to publish it, as follows:

```
CREATE OR REPLACE PROCEDURE
    GREET(str in VARCHAR2)
    AS MLE MODULE jsmodule
    SIGNATURE 'greet(string)';
/
```

The above call specification creates a PL/SQL procedure named `GREET()` in the schema of the current user. Running the procedure executes the exported function `greet()` in the JavaScript module `jsmodule`.

Note that it is not a requirement that the call specification has the same name (`GREET`) as the function being published (`greet`).

The MLE specific clause `MLE MODULE <module name>` specifies the JavaScript MLE module that exports the JavaScript function to be called.

The `SIGNATURE` clause specifies the name of the exported function to be called (in this case, `greet`), as well as, optionally, the list of argument types in parentheses. JavaScript MLE functions use TypeScript types in the `SIGNATURE` clause. In this example, the function accepts a JavaScript string. The PL/SQL `VARCHAR2` string is converted to a JavaScript string before invoking the underlying JavaScript implementation. The `SIGNATURE` clause also allows the list of argument types to be omitted, in which case only the MLE function name is expected and MLE language types are inferred from the types given in the call specification's argument list.

The other exported function, `concat()`, can similarly be used to create a PL/SQL function:

```
CREATE OR REPLACE FUNCTION CONCATENATE(str1 in VARCHAR2, str2 in VARCHAR2)
    RETURN VARCHAR2
    AS MLE MODULE jsmodule
    SIGNATURE 'concat(string, string)';
/
```

The call specification in this case additionally specifies the PL/SQL return type of the created function. The value returned by the JavaScript function `concat()` (of type string) is converted to the type `VARCHAR2`.

The created procedure and function can be called as shown below with the result:

```
SQL> CALL GREET('Peter');
Hello, Peter

Call completed.

SQL> SELECT CONCATENATE('Hello, ','World!');

CONCATENATE('HELLO','WORLD!')
-------------------------------------------------
Hello, World!
```

Topics

- Components of an MLE Call Specification
  The elements of an MLE call specification are listed along with descriptions.
- MLE Module Clause
  The `MLE MODULE` clause specifies the MLE module that exports the underlying JavaScript function for the call specification. The specified module must always be in the same schema as the call specification being created.
- ENV Clause
  The optional `ENV` clause specifies the MLE environment for module contexts in which this call specification will be executed.
- SIGNATURE Clause
  The `SIGNATURE` clause contains all the information necessary to map the MLE call specification to a particular function exported by the specified MLE module.

Components of an MLE Call Specification

The elements of an MLE call specification are listed along with descriptions.

Figure 6-1 MLE Call Specification Syntax

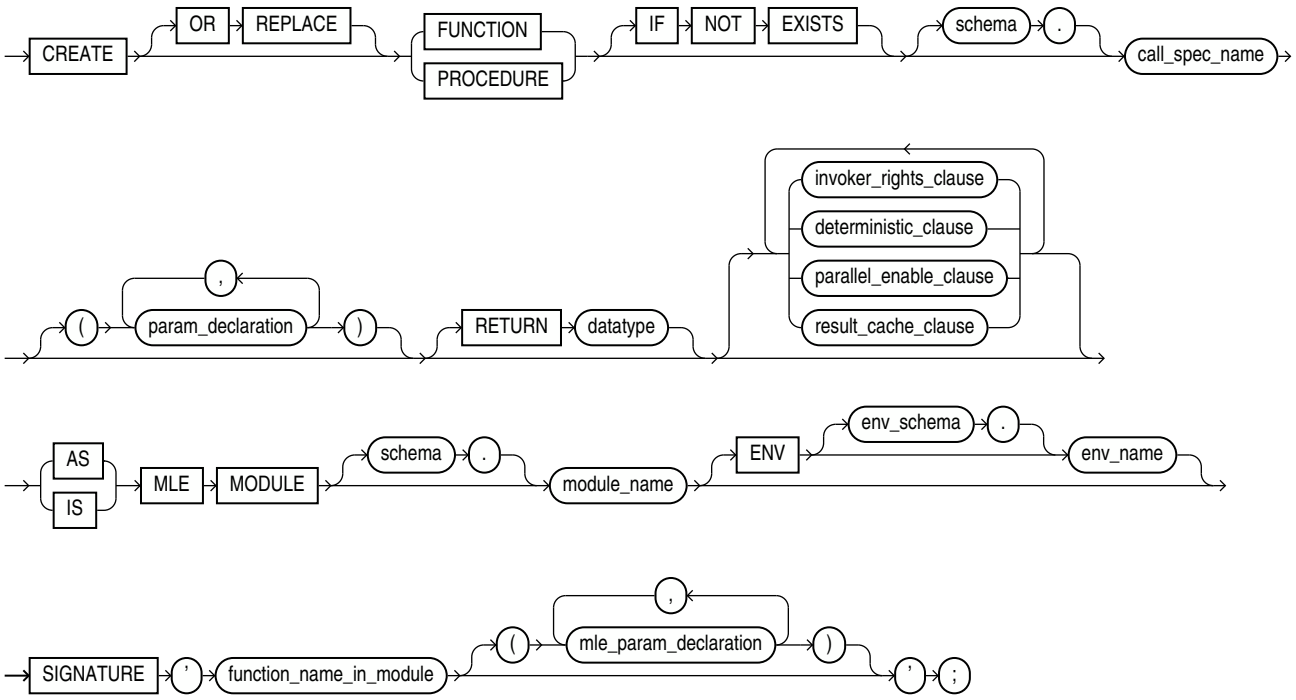


Table 6-1 Components of an MLE Call Specification

Element Name	Description
`OR REPLACE`	Specifies that the function should be replaced if it already exists. This clause can be used to change the definition of an existing function without dropping, recreating, and re-granting object privileges previously granted on the function. Users who had previously been granted privileges on a recreated function or procedure can still access the function without being re-granted the privileges.
`IF NOT EXISTS`	Specifies that the function should be created if it does not already exist. If a function by the same name does exist, the statement is ignored without error and the original function body remains unchanged. Note that SQL*Plus will display the same output message regardless of whether the command is ignored or executed, ensuring that your DDL scripts remain idempotent. `IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.
`schema`	Specifies the schema that will contain the call specification. If the schema is omitted, the call specification is created in the schema of the current user.

Table 6-1 (Cont.) Components of an MLE Call Specification

Element Name	Description
`call_spec_name`	Specifies the name of the call specification to be created. Call specifications are created in the default namespace, unlike MLE modules and environments, which use dedicated namespaces.
`param_declaration`	Specifies the call specification's parameters. If no parameters are specified, parentheses must be omitted.
`RETURN datatype`	Only used for functions and specifies the data type of the return value of the function. The return value can have any data type supported by MLE. Only the data type is specified; length, precision, or scale information must be omitted.
`invoker_rights_clause`	Specifies whether a function is invoker's or definer's rights. • `AUTHID CURRENT_USER` creates an invoker's rights function or procedure. • `AUTHID DEFINER` creates a definer's rights function or procedure. If the `AUTHID` clause is omitted, the call specification is created with definer's rights by default. The `AUTHID` clause on MLE call specifications has the exact same semantics as on PL/SQL functions and procedures.
`deterministic_clause`	Only used for functions and indicates that the function returns the same result value whenever it is called with the same values for its parameters. As with PL/SQL functions, this clause should not be used for functions that access the database in any way that might affect the return result of the function. The results of doing so will not be captured if the database chooses not to re-execute the function.

MLE Module Clause

The `MLE MODULE` clause specifies the MLE module that exports the underlying JavaScript function for the call specification. The specified module must always be in the same schema as the call specification being created.

An `ORA-04103` error is thrown if the specified MLE module does not exist. Likewise, an `ORA-01031` error is raised if the specified module is in a different schema from the created call specification.

ENV Clause

The optional `ENV` clause specifies the MLE environment for module contexts in which this call specification will be executed.

An `ORA-04105` error is thrown if the specified environment schema object does not exist.

If this clause is omitted, the default environment is used. The default environment is simply an environment in its most basic state, with no module imports and no specified language options.

SIGNATURE Clause

The `SIGNATURE` clause contains all the information necessary to map the MLE call specification to a particular function exported by the specified MLE module.

Specifically, it includes two pieces of information:

- The name of the exported function in the specified MLE module.
- The MLE language parameter types (as opposed to the PL/SQL parameter types) for the function (Optional).

The `SIGNATURE` clause must be in the following form:

Figure 6-2 signature_clause ::=

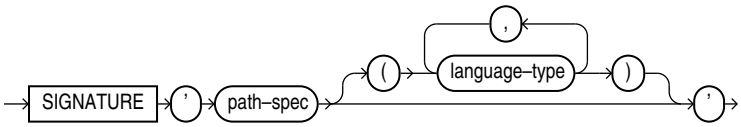


Figure 6-3 path_spec ::=

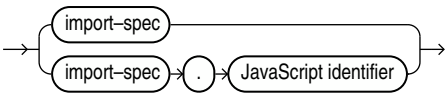
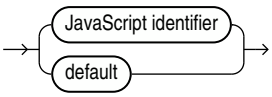


Figure 6-4 import_spec ::=



The path specification describes the function to be called and can have the following two forms:

- A path specification can consist only of an import specification.
  - An import specification can be a JavaScript identifier that identifies a named export of the module, which must be a function. Alternatively, an import specification can be the reserved word, `default`. In this case, the default export of the module is used, which must be a function.
- A path specification can be a composite form consisting of an import specification, followed by a dot and a JavaScript identifier.
  - In this case, the import specification must refer to an object that has a property whose name matches the identifier listed after the dot. The value of the property needs to be a function.

The `language-type` can either be a built-in JavaScript type (e.g. string or number) or a type provided by MLE (e.g. `OracleNumber` or `OracleDate`) that is compatible with the corresponding PL/SQL argument. Note that JSON data maps to the MLE `ANY` type. For an example covering how to pass JSON from PL/SQL to MLE, see Working with JSON Data. For more information about what types are provided by MLE through the built-in module `mle-js-plsqltypes`, see Server-Side JavaScript API Documentation.

`function-name` can include any alphanumeric characters as well as underscores and periods.

When the call specification is a function, the type of the return value is not specified in the `SIGNATURE` clause. Rather, the function can return any JavaScript type that is compatible with the PL/SQL type specified in the call specification's `RETURN` clause.

> **Note:**
>
> The parsing and resolution of the `SIGNATURE` clause happens lazily when the MLE function is executed for the first time. It is only at this point that any resolution or syntax errors in the `SIGNATURE` clause are reported, and not when the call specification is created.

Simplified `SIGNATURE` Clause

`CREATE FUNCTION` and `CREATE PROCEDURE` DDL statements also accept a simplified form of the `SIGNATURE` clause that only specifies the name of the exported function and leaves out the JavaScript language types of the parameters. The default PL/SQL-MLE language type mappings are used in this case.

This example demonstrates the creation of a call specification with a simplified `SIGNATURE` clause.

```
CREATE OR REPLACE FUNCTION concat
    RETURN VARCHAR2
    AS MLE MODULE jsmodule
    SIGNATURE 'concat';
/
```

When the function `concat` is called from PL/SQL, the input `VARCHAR2` parameters are converted to JavaScript string (the default type mapping for `VARCHAR2`) before calling the underlying JavaScript function.

> **See Also:**
>
> MLE Type Conversions for more information about type mappings

Creating an Inline MLE Call Specification

Inline MLE call specifications embed JavaScript code directly in the `CREATE FUNCTION` and `CREATE PROCEDURE` DDLs.

If you want to quickly implement simple functionality using JavaScript, inline MLE call specifications can be a good choice. With this option, you don't need to deploy a separate module containing the JavaScript code. Rather, the JavaScript function is built into the definition of the call specification itself.

The `MLE LANGUAGE` clause is used to specify that the function is implemented using JavaScript. The JavaScript function body must be enclosed by a set of delimiters. Double curly braces are commonly used for this purpose, however, you also have the option to choose your own. The beginning and ending delimiter must match and they cannot be reserved words or a dot. For delimiters such as `{{...}}`, `<<...>>`, and `((...))`, the ending delimiter is the corresponding closing symbol, not an exact match.

The string following the language name is treated as the body of a JavaScript function that implements the functionality of the call specification. When the code is executed, PL/SQL parameters are automatically converted to the default JavaScript type and passed to the

JavaScript function as parameters of the same name. Note that unquoted parameter names are mapped to all-uppercase JavaScript names. The value returned by a JavaScript function is converted to the return type of the PL/SQL call specification, just as with call specifications for MLE modules.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_MLE` subprograms for inline call specifications

Example 6-2 Simple Inline MLE Call Specification

```
CREATE OR REPLACE FUNCTION date_to_epoch (
  "theDate" TIMESTAMP WITH TIME ZONE
)
RETURN NUMBER
AS MLE LANGUAGE JAVASCRIPT
{{
  const d = new Date(theDate);

  //check if the input parameter turns out to be an invalid date
  if (isNaN(d)){
    throw new Error(`${theDate} is not a valid date`);
  }

  //Date.prototype.getTime() returns the number of milliseconds
  //for a given date since epoch, which is defined as midnight
  //on January 1, 1970, UTC
  return d.getTime();
}};
/
```

You can call the function created in the preceding inline call specification using the following SQL statement:

```
SELECT
  date_to_epoch(
    TO_TIMESTAMP_TZ(
      '29.02.2024 11.34.22 -05:00',
      'dd.mm.yyyy hh24:mi:ss tzh:tzm'
    )
  ) epoch_date;
```

Result:

```
EPOCH_DATE
----------
1.7092E+12
```

Example 6-3 Inline MLE Call Specification Returning JSON

Note the use of double quotation marks in the function parameter name, `strArgs`, in Example 6-2. The inclusion preserves the parameter's letter case. Without quotation marks, the parameter name is mapped to an all-uppercase JavaScript name, as seen in this example.

```
CREATE OR REPLACE FUNCTION p_string_to_json(inputString VARCHAR2) RETURN JSON
AS MLE LANGUAGE JAVASCRIPT
{{
  if ( INPUTSTRING === undefined ) {
    throw `must provide a string in the form of key1=value1;...;keyN=valueN`;
  }

  let myObject = {};
  if ( INPUTSTRING.length === 0 ) {
    return myObject;
  }

  const kvPairs = INPUTSTRING.split(";");
  kvPairs.forEach( pair => {
    const tuple = pair.split("=");
    if ( tuple.length === 1 ) {
      tuple[1] = false;
    } else if ( tuple.length != 2 ) {
      throw "parse error: you need to use exactly one '=' between key and
value and not use '=' in either key or value";
    }
    myObject[tuple[0]] = tuple[1];
  });

  return myObject;
}};
/
```

The function created in the preceding inline call specification can be called using the following SQL statement:

```
SELECT p_string_to_json('Hello=Greeting');
```

Result:

```
P_STRING_TO_JSON('HELLO=GREETING')
----------------------------------------------------------------
{"Hello":"Greeting"}
```

- Components of an Inline MLE Call Specification
  The elements of an inline MLE call specification are listed along with descriptions.
- Accessing Built-in Modules Using JavaScript Global Variables
  Rather than importing MLE built-in modules in the same way as call specifications for MLE modules, inline MLE call specifications utilize prepopulated JavaScript globals to access built-in module functionality.

Components of an Inline MLE Call Specification

The elements of an inline MLE call specification are listed along with descriptions.

Figure 6-5 MLE Inline Call Specification Syntax

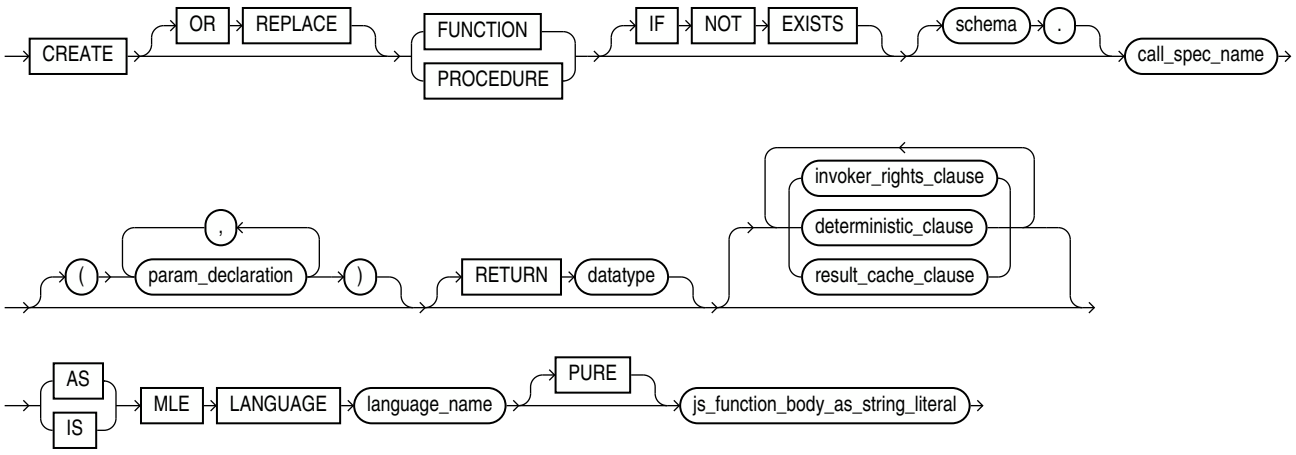


Table 6-2 Components of an Inline MLE Call Specification

Element Name	Description
`OR REPLACE`	Specifies that the function should be replaced if it already exists. This clause can be used to change the definition of an existing function without dropping, recreating, and re-granting object privileges previously granted on the function. Users who had previously been granted privileges on a recreated function or procedure can still access the function without being re-granted the privileges.
`IF NOT EXISTS`	Specifies that the function should be created if it does not already exist. If a function by the same name does exist, the statement is ignored without error and the original function body remains unchanged. Note that SQL*Plus will display the same output message regardless of whether the command is ignored or executed, ensuring that your DDL scripts remain idempotent. `IF NOT EXISTS` cannot be used in combination with `OR REPLACE`.
`schema`	Specifies the schema that will contain the call specification. If the schema is omitted, the call specification is created in the schema of the current user.
`call_spec_name`	Specifies the name of the call specification to be created. Call specifications are created in the default namespace, unlike MLE modules and environments, which use dedicated namespaces.
`param_declaration`	Specifies the call specification's parameters. If no parameters are specified, parentheses must be omitted.
`RETURN datatype`	Only used for functions and specifies the data type of the return value of the function. The return value can have any data type supported by MLE. Only the data type is specified; length, precision, or scale information must be omitted.

Table 6-2 (Cont.) Components of an Inline MLE Call Specification

Element Name	Description
`invoker_rights_clause`	Specifies whether a function is invoker's or definer's rights. • `AUTHID CURRENT_USER` creates an invoker's rights function or procedure. • `AUTHID DEFINER` creates a definer's rights function or procedure. If the `AUTHID` clause is omitted, the call specification is created with definer's rights by default. The `AUTHID` clause on MLE call specifications has the exact same semantics as on PL/SQL functions and procedures.
`deterministic_clause`	Only used for functions and indicates that the function returns the same result value whenever it is called with the same values for its parameters. As with PL/SQL functions, this clause should not be used for functions that access the database in any way that might affect the return result of the function. The results of doing so will not be captured if the database chooses not to re-execute the function.
`MLE LANGUAGE`	Specifies the language of the following code, for example, JavaScript. The string following the language name is interpreted as MLE language code implementing the desired functionality. For JavaScript, this embedded code is interpreted as the body of a JavaScript function.
`PURE`	The `PURE` keyword specifies that the function or procedure should be created in a restricted execution context. During PURE execution, access to database state is disallowed, providing an additional layer of security for user-defined functions that do not require access to database state. For more information, see About Restricted Execution Contexts.

Accessing Built-in Modules Using JavaScript Global Variables

Rather than importing MLE built-in modules in the same way as call specifications for MLE modules, inline MLE call specifications utilize prepopulated JavaScript globals to access built-in module functionality.

Inline MLE call specifications cannot import MLE modules, both built-in and custom. Instead, JavaScript global variables, such as the `session` variable, provide access to the functionality of built-in modules like the JavaScript MLE SQL driver. For more information about the availability of objects in the global scope, see Working with the MLE JavaScript Driver.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the built-in JavaScript modules

Choosing Inline Versus Module MLE Call Specifications

Each option provides its own advantages and disadvantages depending on your use case.

Inline MLE call specifications can simplify the development workflow and provide a way to quickly implement simple JavaScript functionality, as there is no need to deploy a separate module containing the JavaScript code. This is a convenient option if you only need to implement a single JavaScript function. You can use JavaScript global variables to access the functionality of MLE built-in modules but, because inline MLE call specifications are not associated with an MLE environment, modules cannot be imported.

Call specifications for MLE modules offer more flexibility in terms of complexity and ability to import functionality from other modules, built-in and custom. You also have the option to override the default JavaScript type mapping, which is not possible with MLE inline call specifications. Call specifications for MLE modules should be used for larger pieces of JavaScript code as well as for code that you intend to reuse in other JavaScript code using imports.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_MLE` subprograms for MLE call specifications

Runtime Isolation for an MLE Call Specification

MLE uses execution contexts to maintain runtime state isolation. Call specifications are associated with separate contexts when they do not share the same user, module, and environment.
MLE execution contexts act as standalone, isolated runtime environments. All JavaScript code that shares an execution context has full access to all of its runtime state (e.g. any global variables previously defined). Otherwise, there is no way for code executing in one execution context to see or modify runtime state in another execution context. Execution contexts for call specifications are created transparently on the first call to any of the corresponding call specifications. For more information, see About MLE Execution Contexts.

When executing call specifications in a session, MLE loads the module specified in the call specification and calls the function(s) exported by that module. In order for the execution of two call specifications to share the same execution context, they must export a function from the same MLE module, use the same environment, and be executed by the same user. SQL or PL/SQL calls on behalf of different users within the same session are never executed in the same execution context.

The runtime representation of a module is stateful. State constitutes, for example, variables in the JavaScript module as well as variables in the global scope accessible to code in the module. Within the same session, MLE may employ multiple module contexts to execute call specifications. If either the module or the environment referred to by a call specification change, any execution context is invalidated and an error is thrown. Example 6-4 demonstrates this concept.

Session state is very important for data integrity. Not catching errors related to changed session state (`ORA-04106` for module changes and `ORA-04107` for environment changes in JavaScript, as well as `ORA-04068` for PL/SQL packages) can result in silent data corruption.

Setting the initialization parameter `SESSION_EXIT_ON_PACKAGE_STATE_ERROR` to `TRUE` forces sessions to be disconnected if the session state is invalidated. Because many applications capture session disconnect, this option can help simplify the recovery from the invalidation of existing session state. For more information about `SESSION_EXIT_ON_PACKAGE_STATE_ERROR`, see *Oracle Database Reference*.

> **Note:**
>
> Storing state in packages and JavaScript modules is not recommended. Session state is best handled by the database.

All definer's rights call specifications that publish functions from the same MLE module (and use the same environment) will share the same execution context because all execution happens on behalf of the definer. Conversely, there is a separate execution context per calling user when a call specification is declared as invoker's rights.

For more information about how to build a call specification, see Components of an MLE Call Specification.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for information about using `SESSION_EXIT_ON_PACKAGE_STATE_ERROR` to specify behavior when PL/SQL package state is invalidated

Example 6-4 Execution Context Dependencies

This example demonstrates the fact that if a module or environment changes, any associated execution context(s) are invalidated.

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

let myCounter = 0;

export function incrementCounter(){
    return ++myCounter;
}
/

CREATE OR REPLACE FUNCTION increment_and_get_counter
RETURN NUMBER
AS MLE MODULE count_module
SIGNATURE 'incrementCounter';
/
```

Session 1 creates its execution context by invoking the function `increment_and_get_counter`:

```
SQL> SELECT increment_and_get_counter;

INCREMENT_AND_GET_COUNTER
```

```
-------------------------
                        1

SQL> SELECT increment_and_get_counter;

INCREMENT_AND_GET_COUNTER
-------------------------
                        2
```

Another user invoking the function from a different session, we'll say session 2, creates another execution context, separate from the first session's context:

```
SQL> SELECT increment_and_get_counter;

INCREMENT_AND_GET_COUNTER
-------------------------
                        1
```

The user in session 1 recreates the MLE module with some new comments added to the function:

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

let myCounter = 0;

/**
* increments a counter before returning the value
* to the caller
*@returns {number} the value of the counter
*/
export function incrementCounter(){
    return ++myCounter;
}
/
```

This operation signals to all execution contexts referring to `count_module` that their session state should be invalidated. Session 2 gets an error in response to the invalidation:

```
SQL> SELECT increment_and_get_counter;

SELECT increment_and_get_counter
*
ERROR at line 1:
ORA-04106: Module USER2.COUNT_MODULE referred to by INCREMENT_AND_GET_COUNTER
has been modified since the execution context was created.
```

The next invocation of the function in session 2 starts off with a reinitialized session state:

```
SQL> SELECT increment_and_get_counter;

INCREMENT_AND_GET_COUNTER
```

```
-------------------------
                        1
```

Just as with PL/SQL packages, invoking the function from session 1 does not result in an error. Nevertheless, the session state has been discarded as shown by a subsequent call to the function:

```
SQL> SELECT increment_and_get_counter;

INCREMENT_AND_GET_COUNTER
-------------------------
                        1
```

If the initialization parameter `SESSION_EXIT_ON_PACKAGE_STATE_ERROR` is set to `TRUE` in session 2, the `ORA-04106` error is thrown and the connection to the database is cut:

```
ALTER SESSION SET SESSION_EXIT_ON_PACKAGE_STATE_ERROR = TRUE;
SELECT increment_and_get_counter;
```

Result:

```
SELECT increment_and_get_counter
                               *
ERROR at line 1:
ORA-04106: Module USER2.COUNT_MODULE referred to by INCREMENT_AND_GET_COUNTER
has been modified since the execution context was created.

ERROR:
ORA-03114: not connected to ORACLE
```

Dictionary Views for Call Specifications

Metadata about JavaScript call specifications is available in the data dictionary using the `[USER | ALL | DBA | CDB]_MLE_PROCEDURES` views. The family of views maps call specifications (package, function, procedure) to JavaScript modules. This dictionary view is closely modeled after the `*_PROCEDURES` views.

For more information about `*_MLE_PROCEDURES`, see *Oracle Database Reference*.

Example 6-5 Show JavaScript Call Specification Metadata

```
SELECT OBJECT_NAME, PROCEDURE_NAME, SIGNATURE, ENV_NAME, MODULE_NAME
    FROM USER_MLE_PROCEDURES;
```

SQL*Plus output:

```
OBJECT_NAME   PROCEDURE_NAME   SIGNATURE              ENV_NAME    MODULE_NAME
------------  ---------------  ---------------------  ---------
------------
CONCATENATE                    concat(string, string)             JSMODULE
DO_NOTHING                     doNothing(string)                  JSMODULE
```

OUT and IN OUT Parameters

Use `OUT` and `IN OUT` parameters with MLE JavaScript functions.
MLE JavaScript functions support `IN OUT` and `OUT` parameters in addition to `IN` parameters, just as they are supported in PL/SQL functions and procedures. These are declared as `IN OUT` and `OUT` in the list of arguments of an MLE call specification.

Because JavaScript has no notion of output parameters, the JavaScript implementation instead accepts objects that wrap the parameter value. Concretely, the shape of these wrapper objects is described by the following generic TypeScript interfaces `InOut` and `Out` (for `IN OUT` and `OUT` parameters, respectively):

```
Interface InOut<T> {
    Value : T;
}


Interface Out<T> {
    Value : T;
}
```

Note that `OUT` and `IN OUT` parameters are passed to JavaScript functions as JavaScript objects whose only property, `value`, exposes the value of the argument. This means that, in order to read, write, and use the value of an `OUT` or `IN OUT` argument, it must first be unwrapped by accessing its `value` property. This is done in order to simulate a pass-by-reference implementation, which does not exist in JavaScript. For example, the `substitute()` function in Example 6-6 must first unwrap its `IN OUT` argument, `sentence`, by retrieving its `value` property before calling `match()` on it. Attempting to call `match()` on `sentence` directly would fail, as `sentence` is only the value wrapper. These wrapper classes are never needed in `DBMS_MLE`, which does not make use of `OUT` and `IN OUT` parameters.

Example 6-6 OUT and IN OUT Parameters with JavaScript

Consider an MLE function, `substitute()`, that takes a `VARCHAR2 IN OUT` parameter, `sentence`, and replaces all occurrences of the second parameter, `replaceThis`, with the third parameter, `withThat`, then returns the number of occurrences of `replaceThis` in `sentence`.

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

export function substitute (sentence, replaceThis, withThat) {
    /*
     *  substitute: substitutes `replaceThis` in `sentence` with
     *              `replaceThat`
     *
     * parameters:
     * - sentence: the input sentence
     * - replaceThis: a word to be replaced in `sentence`
     * - withThat: the new word to be used instead of `replaceThis`
     */
    const occurrences =
        (sentence.value.match(replaceThis) || []).length;
    sentence.value = sentence.value.replace(replaceThis, withThat);
```

```
    return occurrences;
}
/


CREATE OR REPLACE FUNCTION f_substitute(
    p_sentence       IN OUT VARCHAR2,
    p_replaceThis    IN VARCHAR2,
    p_withThat       IN VARCHAR2
)
RETURN NUMBER
AS MLE MODULE in_out_example_mod
SIGNATURE 'substitute(InOut<string>, string, string)';
/
```

The `SIGNATURE` clause of the call specification lists the parameter type of the JavaScript function's `sentence` parameter as `InOut<string>`. The input `VARCHAR2` value is therefore converted to a JavaScript `string`, that is then wrapped in an object and passed to the JavaScript function `substitute()`.

```
EXEC dbms_session.reset_package
SET SERVEROUTPUT ON

DECLARE
  l_sentence    varchar2(100) := 'people are enjoying the rain';
  l_replaceThis varchar2(100) := 'rain';
  l_withThat    varchar2(100) := 'sun';
  l_occurrences pls_integer;
BEGIN
  dbms_output.put_line('sentence before: ' || l_sentence);
  l_occurrences := f_substitute(
    l_sentence, l_replaceThis, l_withThat);
  if l_occurrences <> 0 then
    dbms_output.put_line('sentence after: ' || l_sentence);
  else
    dbms_output.put_line('no text replacement performed');
  end if;
END;
/
```

Result:

```
sentence before: people are enjoying the rain
sentence after: people are enjoying the sun
```

7

Calling PL/SQL and SQL from the MLE JavaScript SQL Driver

Introduction to the MLE JavaScript SQL Driver

The MLE JavaScript driver is closely modeled after the client-side Oracle SQL driver for Node.js, `node-oracledb`.
This close relationship between the server-side and client-side drivers reduces the effort required to port client-side JavaScript code from Node.js or Deno to the database. Functionality that cannot be reasonably mapped to the server-side environment is omitted from MLE and the MLE JavaScript driver and will throw errors.

This helps you identify those parts of the code requiring changes. Furthermore, the MLE JavaScript implementation is a pure JavaScript implementation. Certain features not part of the ECMAScript standard are unavailable in MLE, such as the window object as well as direct file and network I/O.

The `mle-js-oracledb` SQL driver defaults to a synchronous operating model and partially supports asynchronous execution via async/await.

> **Note:**
>
> Production code should adhere to industry best practices for error handling and logging, which have been omitted from this chapter's examples for the sake of clarity. Additionally, most examples feature the synchronous execution model due to its greater readability.

> **Note:**
>
> If you are running your JavaScript code in a restricted execution context, you cannot use the MLE JavaScript SQL driver. For more information about restricted execution contexts, see About Restricted Execution Contexts.

> **See Also:**
>
> - API Differences Between node-oracledb and mle-js-oracledb
> - Server-Side JavaScript API Documentation for more information about the built-in JavaScript modules

Topics

- Working with the MLE JavaScript Driver
  Generic workflow for working with the MLE JavaScript driver.
- Connection Management in the MLE JavaScript Driver
- Introduction to Executing SQL Statements
- Processing Comparison Between node-oracledb and mle-js-oracledb
  The `node-oracledb` documentation recommends the use of the async/await interface. Due to the nature of client-server interactions, most of the processing involved between node and the database is executed asynchronously.

Working with the MLE JavaScript Driver

Generic workflow for working with the MLE JavaScript driver.

At a high level, working with the MLE JavaScript driver is very similar to using the client-side `node-oracledb` driver, namely:

1. Get a connection handle to the existing database session.
2. Use the connection to execute a SQL statement.
3. Check the result object returned by the statement executed, as well as any database errors that may have occurred.
4. In the case of select statements, iterate over the resulting cursor.
5. For statements manipulating data, decide whether to commit or roll the transaction back.

Applications that aren't ported from client-side Node.js or Deno can benefit from coding aids available in the MLE JavaScript SQL driver, such as many frequently used variables available in the global scope. These variables include the following:

- **oracledb** for the `OracleDb` driver object
- **session** for the default connection object
- **soda** for the `SodaDatabase` object
- **plsffi** for the foreign function interface (FFI) object

Additionally, the following types are available:

- `OracleNumber`
- `OracleClob`
- `OracleBlob`
- `OracleTimestamp`
- `OracleTimestampTZ`
- `OracleDate`
- `OracleIntervalDayToSecond`
- `OracleIntervalYearToMonth`

The availability of these objects in the global scope reduces the need to write boilerplate code. For details about global symbols available with the MLE JavaScript SQL driver, see Server-Side JavaScript API Documentation.

Connection Management in the MLE JavaScript Driver

Considerations when dealing with connection management in the MLE JavaScript driver. Connection management in the MLE JavaScript driver is greatly simplified compared to the client driver. Because a database session will already exist when a JavaScript stored procedure is invoked, you don't need to worry about establishing and tearing down connections, connection pools, and secure credential management, to name just a few.

You need only be concerned with the `getDefaultConnection()` method from the `mle-js-oracledb` module or use the global session object.

Introduction to Executing SQL Statements

A single SQL or PL/SQL statement can be executed by the `Connection` class's `execute()` method. Query results can either be returned in a single JavaScript array or fetched in batches using a `ResultSet` object.
Fetching as `ResultSet` offers more control over the fetch operation whereas using arrays requires fewer lines of code and provides performance benefits unless the amount of data returned is enormous.

Example 7-1 Getting Started with the MLE JavaScript SQL Driver

The following code demonstrates how to import the MLE JavaScript SQL driver into the current module's namespace. This example is based on one provided in the node-oracledb documentation, A SQL SELECT statement in Node.js.

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

import oracledb from "mle-js-oracledb";

/**
```

```
 * Perform a lookup operation on the HR.DEPARTMENTS table to find all
 * departments managed by a given manager ID and print the result on
 * the console
 * @param {number} managerID the manager ID
*/

function queryExample(managerID) {

  if (managerID === undefined) {
    throw new Error (
        "Parameter managerID has not been provided to queryExample()"
    );
  }
  let connection;

  try {
    connection = oracledb.defaultConnection();

    const result = connection.execute(`
        SELECT manager_id, department_id, department_name
        FROM hr.departments
        WHERE manager_id = :id`,
        [
            managerID
        ],
        {
            outFormat: oracledb.OUT_FORMAT_OBJECT
        }
    );
    if (result.rows.length > 0) {
        for (let row of result.rows) {
            console.log(`The query found a row:
                manager_id:      ${row.MANAGER_ID}
                department_id:   ${row.DEPARTMENT_ID}
                department_name: ${row.DEPARTMENT_NAME}`);
        }
    } else {
        console.log(`no data found for manager ID ${managerID}`);
    }

  } catch (err) {
    console.error(`an error occurred while processing the query: $
{err.message}`);
  }
}

export { queryExample };
/
```

The only function present in the module, `queryExample()`, selects a single row from the HR departments table using a bind variable by calling `connection.execute()`. The value of the bind variable is passed as a parameter to the function. Another parameter passed to `connection.execute()` indicates that each row returned by the query should be provided as a JavaScript object.

If data has been found for a given `managerID`, it is printed on the screen. By default, the call to `console.log()` is redirected to `DBMS_OUTPUT`. Should there be no rows returned a message indicating this fact is printed on the console.

The call specification in the following snippet allows the code to be invoked in the database.

```
CREATE OR REPLACE PROCEDURE p_js_sql_query_ex(
    p_manager_id number)
AS MLE MODULE js_sql_mod
SIGNATURE 'queryExample(number)';
/
```

Provided the defaults are still in place, invoking `p_js_sql_query_ex` displays the following:

```
SQL> set serveroutput on
SQL> EXEC p_js_sql_query_ex(103)
The query found a row:
manager_id:      103
department_id:   60
department_name: IT
```

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the built-in JavaScript modules, including `mle-js-oracledb`

Example 7-2 Use Global Variables to Simplify SQL Execution

Example 7-1 can be greatly simplified for use with MLE. Variables injected into the global scope can be referenced, eliminating the need to import the `mle-js-oracledb` module. Additionally, because only a single function is defined in the module, an inline call specification saves even more typing.

```
CREATE OR REPLACE PROCEDURE js_sql_mod_simplified(
    "managerID" number
) AS MLE LANGUAGE JAVASCRIPT
{{
if (managerID === undefined || managerID === null){
    throw new Error (
        "Parameter managerID has not been provided to js_sql_mod_simplified()"
    );
}

const result = session.execute(`
    SELECT
        manager_id,
        department_id,
        department_name
    FROM
        hr.departments
    WHERE
        manager_id = :id`,
```

```
    [ managerID ]
);

if(result.rows.length > 0){
    for(let row of result.rows){
        console.log(
            `The query found a row:
             manager_id: ${row.MANAGER_ID}
             department_id: ${row.DEPARTMENT_ID}
             department_name: ${row.DEPARTMENT_NAME}`
        );
    }
} else {
    console.log(`no data found for manager ID ${managerID}`);
}
}};
/


js_sql_mod_simplified

SQL> set serveroutput on
SQL> exec js_sql_mod_simplified(100);

The query found a row:
manager_id:      100
department_id:   90
department_name: Executive
```

Processing Comparison Between node-oracledb and mle-js-oracledb

The `node-oracledb` documentation recommends the use of the async/await interface. Due to the nature of client-server interactions, most of the processing involved between node and the database is executed asynchronously.

The MLE JavaScript driver does not require asynchronous processing. Like the PL/SQL driver, this is thanks to the driver's location within the database. The MLE JavaScript driver understands the async/await syntax, however, it processes requests synchronously under the hood.

Unlike the `node-oracledb` driver, the MLE JavaScript SQL driver returns rows as objects (`oracledb.OUT_FORMAT_OBJECT`) rather than arrays (`oracledb.OUTFORMAT_ARRAY`) when using the ECMAScript 2023 syntax. Code still relying on the deprecated `require` syntax remains backwards compatible by returning rows as an array.

> **Note:**
>
> A promise-based interface is not provided with the MLE JavaScript driver.

Selecting Data Using the MLE JavaScript Driver

Data can be selected using Direct Fetches or `ResultSet` objects.

You can choose between arrays and objects as the output format. The default is to return data through Direct Fetch using JavaScript objects.

Topics

Direct Fetch: Arrays

Direct Fetches are the default in the MLE JavaScript driver.
Direct Fetches provide query results in `result.rows`. This is a multidimensional JavaScript array if you specify the `outFormat` as `oracledb.OUT_FORMAT_ARRAY`. Iterating over the rows allows you to access columns based on their position in the select statement. Changing the column order in the select statement requires modifications in the parsing of the output. Because this can lead to bugs that are hard to detect, the MLE JavaScript SQL driver returns objects by default (`oracledb.OUT_FORMAT_OBJECT`), rather than arrays.

Example 7-3 demonstrates Direct Fetches using the synchronous execution model.

Example 7-3 Selecting Data Using Direct Fetch: Arrays

```
CREATE OR REPLACE PROCEDURE dir_fetch_arr_proc
AS MLE LANGUAGE JAVASCRIPT
{{
const result = session.execute(
    `SELECT
        department_id,
        department_name
    FROM
        hr.departments
    FETCH FIRST 5 ROWS ONLY`,
    [],
    {
        outFormat: oracledb.OUT_FORMAT_ARRAY
    }
);
for (let row of result.rows) {
    const deptID = String(row[0]).padStart(3, '0');
    const deptName = row[1];
    console.log(`department ID: ${deptID} - department name: ${deptName}`);
}
}};
/

BEGIN
    dir_fetch_arr_proc;
END;
/
```

Result:

```
department ID: 010 - department name: Administration
department ID: 020 - department name: Marketing
department ID: 030 - department name: Purchasing
department ID: 040 - department name: Human Resources
department ID: 050 - department name: Shipping
```

The `execute()` function returns a `result` object. Different properties are available for further processing depending on the statement type (select, insert, delete, etc.).

For information about `mle-js-oracledb`, see Server-Side JavaScript API Documentation.

Direct Fetch: Objects

JavaScript objects are returned by default when using Direct Fetch.
To address potential problems with the ordering of columns in the select list, results are returned as JavaScript objects rather than as arrays.

Example 7-4 Selecting Data Using Direct Fetch: Objects

```
CREATE OR REPLACE PROCEDURE dir_fetch_obj_proc
AS MLE LANGUAGE JAVASCRIPT
{{
const result = session.execute(
    `SELECT
        department_id,
        department_name
    FROM
        hr.departments
    FETCH FIRST 5 ROWS ONLY`,
    [],
    { outFormat: oracledb.OUT_FORMAT_OBJECT }
);

for (let row of result.rows) {
    const deptID = String(row.DEPARTMENT_ID).padStart(3, '0');
    const deptName = row.DEPARTMENT_NAME;
    console.log(`department ID: ${deptID} - department name: ${deptName}`);
}
}};
/

BEGIN
    dir_fetch_obj_proc();
END;
/
```

Result:

```
department ID: 010 - department name: Administration
department ID: 020 - department name: Marketing
department ID: 030 - department name: Purchasing
```

```
department ID: 040 - department name: Human Resources
department ID: 050 - department name: Shipping
```

Unlike PL/SQL, JavaScript doesn't support the concept of named parameters. The `execute()` method accepts the SQL statement, `bindParams`, and options, in that exact order. The query doesn't use bind variables, thus an empty array matches the function's signature.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the `mle-js-oracledb` built-in module

Fetching Rows as ResultSets: Arrays

You can use `ResultSet` objects as an alternative to using Direct Fetches.
In addition to using Direct Fetches, it is possible to use `ResultSet` objects. A `ResultSet` is created when the option property `resultSet` is set to true. `ResultSet` rows can be fetched using `getRow()` or `getRows()`.

Because rows are fetched as JavaScript objects by default instead of as arrays, `outFormat` must be defined as `oracledb.OUT_FORMAT_ARRAY` in order to fetch rows as a `ResultSet`.

Example 7-5 Fetching Rows Using a ResultSet

```
CREATE OR REPLACE PROCEDURE dir_fetch_rs_arr_proc
AS MLE LANGUAGE JAVASCRIPT
{{
const result = session.execute(
    `SELECT
        department_id,
        department_name
    FROM
        hr.departments
    FETCH FIRST 5 ROWS ONLY`,
    [],
    {
        resultSet: true,
        outFormat: oracledb.OUT_FORMAT_ARRAY
    }
);

const rs = result.resultSet;
let row;
while ((row = rs.getRow())){
    const deptID = String(row[0]).padStart(3, '0');
    const deptName = row[1];
    console.log(`department ID: ${deptID} - department name: ${deptName}`);
}
rs.close();
}};
/
```

Note that the fetch operation specifically requested an array rather than an object. Objects are returned by default.

```
EXEC dir_fetch_rs_arr_proc();
```

Result:

```
department ID: 010 - department name: Administration
department ID: 020 - department name: Marketing
department ID: 030 - department name: Purchasing
department ID: 040 - department name: Human Resources
department ID: 050 - department name: Shipping
```

Fetching Rows as ResultSets: Iterating Over ResultSet Objects

In addition to the `ResultSet.getRow()` and `ResultSet.getRows()` functions, the MLE JavaScript driver's `ResultSet` implements the iterable and iterator protocols, simplifying the process for iterating over the `ResultSet`.
Using either the iterable or iterator protocols is possible. Both greatly simplify working with `ResultSets`. The iterable option is demonstrated in Example 7-6.

> **Note:**
>
> `ResultSet` objects must be closed once they are no longer needed.

Example 7-6 Using the Iterable Protocol with ResultSets

This example shows how to use the iterable protocol as an alternative to `ResultSet.getRow()`. Rather than providing an array of column values, the JavaScript objects are returned instead.

```
CREATE OR REPLACE PROCEDURE rs_iterable_proc
AS MLE LANGUAGE JAVASCRIPT
{{
const result = session.execute(
    `SELECT
        department_id,
        department_name
    FROM
        hr.departments
    FETCH FIRST 5 ROWS ONLY`,
    [],
    {
        resultSet: true
    }
);
const rs = result.resultSet;
for (let row of rs){
    const deptID = String(row.DEPARTMENT_ID).padStart(3, '0');
    const deptName = row.DEPARTMENT_NAME;
    console.log(`department ID: ${deptID} - department name: ${deptName}`);
}
rs.close();
```

```
}};
/

BEGIN
    rs_iterable_proc();
END;
/
```

Result:

```
department ID: 010 - department name: Administration
department ID: 020 - department name: Marketing
department ID: 030 - department name: Purchasing
department ID: 040 - department name: Human Resources
department ID: 050 - department name: Shipping
```

Data Modification

Modify data using the MLE JavaScript SQL driver.
In addition to selecting data, it is possible to insert, update, delete, and merge data using the MLE JavaScript SQL driver. The same general workflow can be applied to these operations as you would use when selecting data.

Example 7-7 Updating a Row Using the MLE JavaScript SQL Driver

```
CREATE OR REPLACE MLE MODULE row_update_mod LANGUAGE JAVASCRIPT AS
import oracledb from "mle-js-oracledb";
export function updateCommissionExampleEmpID145() {
    const conn = oracledb.defaultConnection();
    const result = conn.execute(
        `UPDATE employees
         SET commission_pct = commission_pct * 1.1
         WHERE employee_id = 145`
    );
    return result.rowsAffected;
}
/
```

The `result` object's `rowsAffected` property can be interrogated to determine how many rows have been affected by the update. The JavaScript function `updateCommissionExampleEmpID145()` returns the number of rows affected to the caller. In this instance, the function will return `1`.

An alternative method to update data is to use the `connection.executeMany()` method. This function works best when used with bind variables.

Bind Variables

Use bind variables to control data passed into or retrieved from the database.
SQL and PL/SQL statements may contain bind variables, indicated by colon-prefixed identifiers. These parameters indicate where separately specified values are substituted in a statement when executed, or where values are to be returned after execution.

Three different kinds of bind variables exist in the Oracle database:

- `IN` bind variables
- `OUT` bind variables
- `IN OUT` bind variables

`IN` binds are values passed into the database. `OUT` binds are used to retrieve data from the database. `IN OUT` binds are passed in and may return a different value after the statement executes.

Using bind variables is recommended in favor of constructing SQL or PL/SQL statements through string concatenation or template literals. Both performance and security can benefit from the use of bind variables. When bind variables are used, the Oracle database does not have to perform a resource and time consuming hard-parse operation. Instead, it can reuse the cursor already present in the cursor cache.

> **Note:**
>
> Bind variables cannot be used in DDL statements such as `CREATE TABLE`, nor can they substitute the text of a query, only data.

Topics

Using Bind-by-Name vs Bind-by-Position

Bind variables are used in two ways: by name by position. You must pick one for a given SQL command as the options are mutually exclusive.

Topics

Named Bind Variables

Binding by name requires the bind variable to be a string literal, prefixed by a colon.
In the case of named binds, the `bindParams` argument to the `connection.execute()` function should ideally be provided with the following properties of each bind variable defined.

Property	Description
`dir`	The bind variable direction
`val`	The value to be passed to the SQL statement
`type`	The data type

Example 7-8 Using Named Bind Variables

```
CREATE OR REPLACE PROCEDURE named_binds_ex_proc(
    "deptName" VARCHAR2,
    "sal" NUMBER
)
AS MLE LANGUAGE JAVASCRIPT
{{
if (deptName === null || sal === null){
    throw new Error(
        `must provide deptName and sal to named_binds_ex_proc()`
    );
}

const result = session.execute(
    `SELECT
        e.first_name ||
        ''    ||
        e.last_name employee_name,
        e.salary
    FROM
        hr.employees e
        LEFT JOIN hr.departments d ON (e.department_id = d.department_id)
    WHERE
        nvl(d.department_name, 'n/a') = :deptName
        AND salary > :sal
    ORDER BY
        e.employee_id`,
    {
        deptName:{
            dir: oracledb.BIND_IN,
            val: deptName,
            type: oracledb.STRING
        },
        sal:{
            dir: oracledb.BIND_IN,
            val: sal,
            type: oracledb.NUMBER
        }
    }
);
console.log(`Listing employees working in ${deptName} with a salary > $
{sal}`);
for (let row of result.rows){
    console.log(`${row.EMPLOYEE_NAME.padEnd(25)} - ${row.SALARY}`);
}
}};
/
```

The `bindParams` argument to `connection.execute()` defines two named bind parameters:

- `deptName`
- `sal`

In this example, the function's input parameters match the names of the bind variables, which improves readability but isn't a requirement. You can assign bind variable names as long as the mapping in `bindParams` is correct.

Positional Bind Variables

Instead of using named bind parameters, you can alternatively provide bind-variable information as an array.
The number of elements in the array must match the number of bind parameters in the SQL text. Rather than mapping by name, the mapping of bind variable and value is based on the position of the bind variable in the text and position of the item in the bind array.

Example 7-9 Using Positional Bind Variables

This example demonstrates the use of positional bind variables and represents a reimplementation of Example 7-8

```
CREATE OR REPLACE PROCEDURE positional_binds_ex_proc(
    "deptName" VARCHAR2,
    "sal" NUMBER
)
AS MLE LANGUAGE JAVASCRIPT
{{
if (deptName === null || sal === null){
    throw new Error(
        `must provide deptName and sal to positional_binds_ex_proc()`
    );
}

const result = session.execute(
    `SELECT
        e.first_name ||
        ''    ||
        e.last_name employee_name,
        e.salary
    FROM
        hr.employees e
        LEFT JOIN hr.departments d ON (e.department_id = d.department_id)
    WHERE
        nvl(d.department_name, 'n/a') = :deptName
        AND salary > :sal
    ORDER BY
        e.employee_id`,
    [
        deptName,
        sal
    ]
);
console.log(`Listing employees working in ${deptName} with a salary > $
{sal}`);
for(let row of result.rows){
    console.log(`${row.EMPLOYEE_NAME.padEnd(25)} - ${row.SALARY}`);
}
}};
/
```

In this example, `bindParams` is an array rather than an object. The mapping between bind variables in the SQL text to values is done by position. The first item in the `bindParams` array maps to the first occurrence of a placeholder in the SQL text and so on.

RETURNING INTO Clause

The use of the `RETURNING INTO` clause is described.
The `RETURNING INTO` clause allows you to

- Fetch values changed during an update
- Return auto-generated keys during a single-row insert operation
- List rows deleted

Example 7-10 Using the RETURNING INTO Clause

This example shows how to retrieve the old and new values after an update operation. These values can be used for further processing.

```
CREATE OR REPLACE PROCEDURE ret_into_ex_proc(
    "firstEmpID" NUMBER,
    "lastEmpID" NUMBER
)
AS MLE LANGUAGE JAVASCRIPT
{{
if (firstEmpID === null || lastEmpID === null){
    throw new Error(
        `must provide deptName and sal to ret_into_ex_proc()`
    );
}

const result = session.execute(
    `UPDATE
        hr.employees
    SET
        last_name = upper(last_name)
    WHERE
        employee_id between :firstEmpID and :lastEmpID
    RETURNING
        old last_name
        new last_name
    INTO
        :oldLastName,
        :newLastName`,
    {
        firstEmpID: {
            dir: oracledb.BIND_IN,
            val: firstEmpID,
            type: oracledb.NUMBER
        },
        lastEmpID: {
            dir: oracledb.BIND_IN,
            val: lastEmpID,
            type: oracledb.NUMBER
        },
        oldLastName: {
```

```
            type: oracledb.STRING,
            dir: oracledb.BIND_OUT
        },
        newLastName: {
            type: oracledb.STRING,
            dir: oracledb.BIND_OUT
        }
    }
);

if (result.rowsAffected > 1){
    console.log(
        `update() completed successfully:
        - old values: ${JSON.stringify(result.outBinds.oldLastName)}
        - new values: ${JSON.stringify(result.outBinds.newLastName)}`
    );
} else {
    throw new Error(
        `found no row to update in range ${firstEmpID} to ${lastEmpID}`
    );
}
}};
/
```

This example features both `IN` and `OUT` bind variables:

- `firstEmpID` and `lastEmpID` specify the data range to be updated
- `oldLastName` is an array containing all the last names as they were before the update
- `newLastName` is another array containing the new values

Batch Operations

In addition to calling the `connection.execute()` function, it is possible to use `connection.executeMany()` to perform batch operations.
Using `connection.executeMany()` is like calling `connection.execute()` multiple times but requires less work. This is an efficient way to handle batch changes, for example, when inserting or updating multiple rows. The `connection.executeMany()` method cannot be used for queries.

`connection.execute()` expects an array containing variables to process by the SQL statement. The `bindData` array in Example 7-11 contains multiple JavaScript objects, one for each bind variable defined in the SQL statement. The for loop constructs the objects and adds them to the `bindData` array.

In addition to the values to be passed to the batch operation, the MLE JavaScript SQL driver needs to know about the values' data types. This information is passed as the `bindDefs` property in the `connection.executeMany()` options parameter. Both old and new last names in Example 7-11 are character strings with the `changeDate` defined as a date.

Just as with the `connection.execute()` function, `connection.executeMany()` returns the `rowsAffected` property, allowing you to quickly identify how many rows have been batch processed.

Example 7-11 Performing a Batch Operation

This example extends Example 7-9 by inserting the old and new last names into an audit table.

```
CREATE OR REPLACE PROCEDURE ret_into_audit_ex_proc(
    "firstEmpID" NUMBER,
    "lastEmpID" NUMBER
)
AS MLE LANGUAGE JAVASCRIPT
{{
if (firstEmpID === null || lastEmpID === null){
    throw new Error(
        `must provide deptName and sal to ret_into_audit_ex_proc()`
    );
}

let result = session.execute(
    `UPDATE
        hr.employees
    SET
        last_name = upper(last_name)
    WHERE
        employee_id between :firstEmpID and :lastEmpID
    RETURNING
        old last_name,
        new last_name
    INTO
        :oldLastName,
        :newLastName`,
    {
        firstEmpID: {
            dir: oracledb.BIND_IN,
            val: firstEmpID,
            type: oracledb.NUMBER
        },
        lastEmpID: {
            dir: oracledb.BIND_IN,
            val: lastEmpID,
            type: oracledb.NUMBER
        },
        oldLastName: {
            type: oracledb.STRING,
            dir: oracledb.BIND_OUT
        };
        newLastName: {
            type: oracledb.STRING,
            dir: oracledb.BIND_OUT
        }
    }
);

if (result.rowsAffected > 1){
    // store the old data and new values in an audit table
    let bindData = [];
    const changeDate = new Date();
    for (let i = 0; i < result.outBinds.oldLastName.length, i++){
```

```
            bindDate.push(
                {
                    oldLastName: result.outBinds.oldLastName[i],
                    newLastName: result.outBinds.newLastName[i],
                    changeDate: changeDate
                }
            );
        }
        // use executeMany() with the newly populated array
        result = session.executeMany(
            `insert into EMPLOYEES_AUDIT_OPERATIONS(
                old_last_name,
                new_last_name,
                change_date
            ) values (
                :oldLastName,
                :newLastName,
                :changeDate
            )`,
            bindData,
            {
                bindDefs: {
                    oldLastName: {type: oracledb.STRING, maxSize: 30},
                    newLastName: {type: oracledb.STRING, maxSize: 30},
                    changeDate: {type: oracledb.DATE}
                }
            }
        );

    } else {
        throw new Error(
            `found no row to update in range ${firstEmpID} to ${lastEmpID}`
        );
    }
    }};
    /
```

After the initial update statement completes, the database provides the old and new values of the `last_name` column affected by the update in the `result` object's `outBinds` property. Both `oldLastName` and `newLastName` are arrays. The array length represents the number of rows updated.

PL/SQL Invocation from the MLE JavaScript SQL Driver

Use the MLE JavaScript driver to call functions and procedures from PL/SQL.
Most of the Oracle Database's API is provided in PL/SQL. This is not a problem; you can easily call PL/SQL from JavaScript. Invoking PL/SQL using the MLE JavaScript SQL driver is similar to calling SQL statements.

Example 7-12 Calling PL/SQL from JavaScript

```
CREATE OR REPLACE MLE MODULE plsql_js_mod
LANGUAGE JAVASCRIPT AS
/**
 * Read the current values for module and action and return them as
```

```
 * a JavaScript object. Typically set before processing starts to
 * allow you to restore the values if needed.
 * @returns an object containing module and action
 */
function preserveModuleAction(){
    //Preserve old module and action. DBMS_APPLICATION_INFO provides
    // current module and action as OUT binds
    let result = session.execute(
        `BEGIN
            DBMS_APPLICATION_INFO.READ_MODULE(
                :l_module,
                :l_action
            );
        END;`,
        {
            l_module: {
                dir: oracledb.BIND_OUT,
                type: oracledb.STRING
            },
            l_action: {
                dir: oracledb.BIND_OUT,
                type: oracledb.STRING
            }
        }
    );

    // Their value can be assigned to JavaScript variables
    const currentModule = result.outBinds.l_module;
    const currentAction = result.outBinds.l_action;

    // ... and returned to the caller
    return {
        module: currentModule,
        action: currentAction
    }
}

/**
 * Set module and action using DBMS_APPLICATION_INFO
 * @param theModule the module name to set
 * @param theAction the name of the action to set
 */
function setModuleAction(theModule, theAction){
    session.execute(
        `BEGIN
            DBMS_APPLICATION_INFO.SET_MODULE(
                :module,
                :action
            );
        END;`,
        [
            theModule,
            theAction
        ]
    );
}
```

```
/**
 * The only public function in this module simulates some heavy
 * processing for which module and action are set using the built-in
 * DBMS_APPLICATION_INFO package.
 */
export function plsqlExample(){
    // preserve the values for module and action before we begin
    const moduleAction = preserveModuleAction();

    // set the new values to reflect the function's execution
    // within the module
    setModuleAction(
        'plsql_js_mod',
        'plsqlExample()'
    )

    // Simulate some intensive processing... While this is ongoing
    // module and action in v$session should have changed to the
    // values set earlier. You can check using
    // SELECT module, action FROM v$session WHERE module = 'plsql_js_mod'
    session.execute(
        `BEGIN
            DBMS_SESSION.SLEEP(60);
        END;`
    );

    // and finally reset the values to what they were before
    setModuleAction(
        moduleAction.module,
        moduleAction.action
    );
}
/
```

This example is a little more elaborate than previous ones, separating common functionality into their own (private) functions. You can see the use of `OUT` variables in `preserveModuleAction()`'s call to `DBMS_APPLICATION_INFO`. The values can be retrieved using `result.outBinds`.

After storing the current values of module and action in local variables, additional anonymous PL/SQL blocks are invoked, first setting module and action before entering a 60-second sleep cycle simulating complex data processing. Once the simulated data processing routine finishes, the module and action are reset to their original values using named `IN` bind variables. Using bind variables is more secure than string concatenation.

Setting module and action is an excellent way of informing the database about ongoing activity and allows for better activity grouping in performance reports.

Error Handling in SQL Statements

JavaScript provides an exception framework like Java. Rather than returning an `Error` object as a promise or callback as in `node-oracledb`, the MLE JavaScript driver resorts to throwing errors. This concept is very familiar to PL/SQL developers.

Using try-catch-finally in JavaScript code is similar to the way PL/SQL developers use begin-exception-end blocks to trap errors during processing.

Use the JavaScript `throw()` command if an exception should be re-thrown. This causes the error to bubble-up the stack after it has been dealt with in the catch block. Example 7-14 demonstrates this concept.

Example 7-13 SQL Error Handling Inside a JavaScript Function

```
CREATE TABLE log_t (
    id NUMBER GENERATED ALWAYS AS IDENTITY
    CONSTRAINT pk_log_t PRIMARY KEY,
    err VARCHAR2(255),
    msg VARCHAR2(255)
);

CREATE OR REPLACE PACKAGE logging_pkg as
  PROCEDURE log_err(p_msg VARCHAR2, p_err VARCHAR2);
END logging_pkg;
/

CREATE OR REPLACE PACKAGE BODY logging_pkg AS
  PROCEDURE log_err(p_msg VARCHAR2, p_err VARCHAR2)
  AS
    PRAGMA autonomous_transaction;
  BEGIN
    INSERT INTO log_t (
        err,
        msg
    ) VALUES (
        p_err,
        p_msg
    );
    COMMIT;
  END log_err;
END logging_pkg;
/

CREATE OR REPLACE MLE MODULE js_err_handle_mod
LANGUAGE JAVASCRIPT AS

/**
 *short demo showing how to use try/catch to catch an error
 *and proceeding normally. In the example, the error is
 *provoked
*/
export function errorHandlingDemo(){

    try{
        const result = session.execute(
            `INSERT INTO
                surelyThisTableDoesNotExist
            VALUES
                (1)`
        );
```

```
    console.log(`there were ${result.rowsAffected} rows inserted`);

    } catch(err) {
        logError('this is some message', err);

        //tell the caller that something went wrong
        return false;
    }

    //further processing

    //return successful completion of the code
    return true;
}

/**
 *log an error using the logging_pkg created at the beginning
 *of this example. Think of it as a package logging errors in
 *a framework for later analysis.
 *@param msg an accompanying message
 *@param err the error encountered
*/
function logError(msg, err){
    const result = session.execute(
        `BEGIN
            logging_pkg.log_err(
                p_msg => :msg,
                p_err => :err
            );
        END;`,
        {
            msg: {
                val: msg,
                dir: oracledb.BIND_IN
            },
            err: {
                val: err.message,
                dir: oracledb.BIND_IN
            }
        }
    );
}
/
```

Create a function, `js_err_handle_mod_f`, using the module `js_err_handle_mod` as follows:

```
CREATE OR REPLACE FUNCTION js_err_handle_mod_f
RETURN BOOLEAN
AS MLE MODULE js_err_handle_mod
SIGNATURE 'errorHandlingDemo()';
/
```

Now you can call the function and use the return value to see whether the processing was successful:

```
DECLARE
    l_success boolean := false;
BEGIN
    l_success := js_err_handle_mod_f;

    IF l_success THEN
        DBMS_OUTPUT.PUT_LINE('normal, successful completion');
    ELSE
        DBMS_OUTPUT.PUT_LINE('an error has occurred');
    END IF;
END;
/
```

In this case, the error is caught within the MLE module. The error is recorded by the application, allowing the administrator to assess the situation and take corrective action.

Example 7-14 Error Handling Using JavaScript throw() Command

This example demonstrates the use of the JavaScript `throw()` command in the catch block. Unlike the screen output shown for `js_err_handle_mod` in Example 7-13, a calling PL/SQL block will have to catch the error and either treat it accordingly or raise it again.

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

/**
 *a similar example as Example 7-13, however, rather than
 *processing the error in the JavaScript code, it is re-thrown up the call
stack.
 *It is now up to the called to handle the exception. The try/catch block is
not
 *strictly necessary but is used in this example as a cleanup step to remove
Global
 *Temporary Tables (GTTs) and other temporary objects that are no longer
required.
*/
export function rethrowError(){

    try{
        const result = session.execute(
            `INSERT INTO
                surelyThisTableDoesNotExist
            VALUES
                (1)`
        );

        console.log(`there were ${result.rowsAffected} rows inserted`);

    } catch(err){
        cleanUpBatch();

        throw(err);
    }
```

```
    //further processing
}

function cleanUpBatch(){
    //batch cleanup operations
    return;
}
/
```

Using the following call specification, failing to catch the error will result in an unexpected error, which can propagate up the call stack all the way to the end user.

```
CREATE OR REPLACE PROCEDURE rethrow_err_proc
AS MLE MODULE js_throw_mod
SIGNATURE 'rethrowError()';
/

BEGIN
    rethrow_err_proc;
END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-00942: table or view does not exist
ORA-04171: at rethrowError (USER1.JS_THROW_MOD:11:24)
ORA-06512: at "USER1.RETHROW_ERROR_PROC", line 1
ORA-06512: at line 2
```

End users should not see this type of error. Instead, a more user-friendly message should be displayed. Continuing the example, a simple fix is to add an exception block:

```
BEGIN
    rethrow_err_proc;
EXCEPTION
    WHEN OTHERS THEN
        logging_pkg.log_err(
            'something went wrong',
            sqlerrm
        );
        --this would be shown on the user interface;
        --for the sake of demonstration this workaround
        --is used to show the concept
        DBMS_OUTPUT.PUT_LINE(
            'ERROR: the process encountered an unexpected error'
        );
        DBMS_OUTPUT.PUT_LINE(
            'please inform the administrator referring to application error
1234'
        );
```

```
END;
/
```

Result:

```
ERROR: the process encountered an unexpected error
please inform the administrator referring to application error 1234

PL/SQL procedure successfully completed.
```

Working with JSON Data

The use of JSON data as part of a relational structure, more specifically the use of JSON columns in (relational) tables, is described.

Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views. Unlike relational data, JSON data can be stored in the database, indexed, and queried without any need for a schema.[1]

Oracle also provides a family of Simple Oracle Document Access (SODA) APIs for access to JSON data stored in the database. SODA is designed for schemaless application development without knowledge of relational database features or languages such as SQL and PL/SQL. It lets you create and store collections of documents in Oracle Database, retrieve them, and query them without needing to know how the documents are stored in the database.

JSON data is widely used for exchanging information between the application tier and the database. Oracle REST Data Services (ORDS) is the most convenient tool for making REST calls to the database. Example 7-15 demonstrates this concept.

Manipulating JSON is one of JavaScript's core capabilities. Incoming JSON documents don't require parsing using `JSON.parse()`, they can be used straight away. Micro-service architectures greatly benefit from the enhanced options offered by JavaScript in the database.

See Also:

- Working with SODA Collections in MLE JavaScript Code for a detailed discussion of SODA and JavaScript in the database
- *Oracle Database JSON Developer's Guide* for information about the use of JSON in Oracle Database

Example 7-15 Inserting JSON Data into a Database Table

This example assumes that a REST API has been published in ORDS, allowing users to POST JSON data to the database. This way, administrators have the option to upload further

[1] A JSON schema is not to be confused with the concept of a database schema: a database schema in Oracle Database is a separate namespace for database users to create objects such as tables, indexes, views, and many others without risking naming collisions.

departments into the `departments` table. Once the JSON data has been received, the MLE module uses `JSON_TABLE()` to convert the JSON data structure into a relational model.

```
CREATE TABLE departments(
    department_id NUMBER NOT NULL PRIMARY KEY,
    department_name VARCHAR2(50) NOT NULL,
    manager_id NUMBER,
    location_id NUMBER
);

CREATE OR REPLACE FUNCTION REST_API_DEMO(
    "depts" JSON
) RETURN BOOLEAN
AS MLE LANGUAGE JAVASCRIPT
{{
    /**
    *insert a number of department records, provided as JSON,
    *into the departments table
    *@params {object} depts - an array of departments
    */

    if(depts.constructor !== Array){
        throw new Error('must provide an array of departments to this
function');
    }

    //convert JSON input to relational data and insert into a table
    const result = session.execute(`
        INSERT INTO departments(
            department_id,
            department_name,
            manager_id,
            location_id
        )
        SELECT
            jt.*
        FROM json_table(:depts, '$[*]' columns
            department_id    path    '$.department_id',
            department_name  path    '$.department_name',
            manager_id       path    '$.manager_id',
            location_id      path    '$.location_id'
        ) jt`,
        {
            depts:{
                val: depts,
                type: oracledb.DB_TYPE_JSON
            }
        }
    );

    if(result.rowsAffected !== depts.length){
        return false;
    } else {
        return true;
    }
```

```
}};
/
```

Using the following anonymous PL/SQL block to simulate the REST call, additional departments can be inserted into the table:

```
DECLARE
    l_success boolean := false;
    l_depts JSON;
BEGIN
    l_depts := JSON('[
        {
            "department_id": 1010,
            "department_name": "New Department 1010",
            "manager_id": 200,
            "location_id": 1700
        },
        {
            "department_id": 1020,
            "department_name": "New Department 1020",
            "manager_id": 201,
            "location_id": 1800
        },
        {
            "department_id": 1030,
            "department_name": "New Department 1030",
            "manager_id": 114,
            "location_id": 1700
        },
        {
            "department_id": 1040,
            "department_name": "New Department 1040",
            "manager_id": 203,
            "location_id": 2400
        }]'
    );

    l_success := REST_API_DEMO(l_depts);

    IF NOT l_success THEN
        RAISE_APPLICATION_ERROR(
            -20001,
            'an unexpected error occurred ' || sqlerrm
        );
    END IF;
END;
/
```

The data has been inserted successfully as demonstrated by the following query:

```
SELECT *
FROM departments
WHERE department_id > 1000;
```

Result:

```
DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
         1010 New Department 1010                   200        1700
         1020 New Department 1020                   201        1800
         1030 New Department 1030                   114        1700
         1040 New Department 1040                   203        2400
```

Example 7-16 Use JavaScript to Manipulate JSON Data

Rather than using SQL functions like JSON_TABLE, JSON_TRANSFORM, and so on, it is possible to perform JSON data manipulation in JavaScript as well.

This example is based on the `J_PURCHASEORDER` table as defined in *Oracle Database JSON Developer's Guide*. This table stores a JSON document containing purchase orders from multiple customers. Each purchase order consists of one or more line items.

The following function, `addFreeItem()`, allows the addition of a free item to customers ordering merchandise in excess of a threshold value.

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

/**
 *a simple function accepting a purchase order and checking whether
 *its value is high enough to merit the addition of a free item
 *
 *@param {object} po the purchase order to be checked
 *@param {object} freeItem which free item to add to the order free of charge
 *@param {number} threshold the minimum order value before a free item can be
added
 *@param {boolean} itemAdded a flag indicating whether the free item was
successfully added
 *@returns {object} the potentially updated purchaseOrder
 *@throws exception in case
 *    -any of the mandatory parameters is null
 *    -in the absence of line items
 *    -if the free item has already been added to the order
 */
export function addFreeItem(po, freeItem, threshold, itemAdded){

    //ensure values for parameters have been provided
    if(po == null || freeItem == null || threshold == null){
        throw new Error(`mandatory parameter either not provided or null`);
    }

    //make sure there are line items provided by the purchase order
    if(po.LineItems === undefined) {
        throw new Error(
            `PO number ${po.PONumber} does not contain any line items`
        );
    }

    //bail out if the free item has already been added to the purchase order
    if(po.LineItems.find(({Part}) => Part.Description ===
```

```
freeItem.Part.Description)){
        throw new Error(`${freeItem.Part.Description} has already been added
to order ${po.PONumber}`);
    }

    //In, Out, and InOut Parameters are implemented in JavaScript using
    //special interfaces
    itemAdded.value = false;

    //get the total order value
    const poValue = po.LineItems
        .map(x => x.Part.UnitPrice * c.Quantity)
        .reduce(
            (accumulator, currentValue) => accumulator + currentValue, 0
        );

    //add a free item to the purchase order if its value exceeds
    //the threshold
    if(poValue > threshold){

        //update the ItemNumber
        freeItem.ItemNumber = (po.LineItems.length + 1)
        po.LineItems.push(freeItem);
        itemAdded.value = true;
    }

    return po;
}
/
```

As with every MLE module, you must create a call specification before you can use it in SQL and PL/SQL. The following example wraps the call to `add_free_item()` into a package. The function accepts a number of parameters, including an `OUT` parameter, requiring an extended signature clause mapping the PL/SQL types to MLE types. The SQL data type JSON maps to the MLE `ANY` type. Because there is no concept of an `OUT` parameter in JavaScript, the final parameter, `p_item_added`, must be passed using the Out interface. For a more detailed discussion about using bind parameters with JavaScript, see OUT and IN OUT Parameters.

```
CREATE OR REPLACE PACKAGE purchase_order_pkg AS

  FUNCTION add_free_item(
    p_po            IN JSON,
    p_free_item     IN JSON,
    p_threshold     IN NUMBER,
    p_item_added    OUT BOOLEAN
  )
  RETURN JSON AS
  MLE MODULE purchase_order_mod
  SIGNATURE 'addFreeItem(any, any, number, Out<boolean>)';

  --additional code

END purchase_order_pkg;
/
```

Using Large Objects (LOB) with MLE

A PL/SQL wrapper type is used to handle CLOBs and BLOBs with the MLE JavaScript driver. Handling large objects such as CLOBs (Character Large Object) and BLOBs (Binary Large Object) with the MLE JavaScript driver differs from the `node-oracledb` driver. Rather than using a Node.js Stream interface, a PL/SQL wrapper type is used. The wrapper types for BLOBs and CLOBs are called `OracleBlob` and `OracleClob`, respectively. They are defined in `mle-js-plsqltypes`. Most types are exposed in the global scope and can be referenced without having to import the module.

> **Note:**
>
> `BFILE`, commonly counted among LOBs, is not supported.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about `mle-js-plsqltypes` and the other JavaScript built-in modules

Topics

- Writing LOBs
  An example shows how to initialize and write to a CLOB that is finally inserted into a table.
- Reading LOBs
  An example is used to show how to select a CLOB and then use the `fetchInfo` property to read the contents of the CLOB as a string.

Writing LOBs

An example shows how to initialize and write to a CLOB that is finally inserted into a table.

Example 7-17 Inserting a CLOB into a Table

This example demonstrates how to insert a CLOB into a table. The table defines two columns: an ID column to be used as a primary key and a CLOB column named "C".

```
CREATE TABLE mle_lob_example (
    id NUMBER GENERATED ALWAYS AS IDENTITY,
    CONSTRAINT pk_mle_blob_table PRIMARY KEY(id),
    c  CLOB
);

CREATE OR REPLACE PROCEDURE insert_clob
AS MLE LANGUAGE JAVASCRIPT
{{
//OracleClob is exposed in the global scope and does not require
//importing 'mle-js-plsqltypes', similar to how oracledb is available
let theClob = OracleClob.createTemporary(false);
```

```
theClob.open(OracleClob.LOB_READWRITE);
theClob.write(
    1,
    'This is a CLOB and it has been inserted by the MLE JavaScript SQL Driver'
);

const result = session.execute(
    `INSERT INTO mle_lob_example(c) VALUES(:theCLOB)`,
    {
        theCLOB:{
            type: oracledb.ORACLE_CLOB,
            dir: oracledb.BIND_IN,
            val: theCLOB
        }
    }
);

//it is best practice to close the handle to free memory
theCLOB.close();
}};
/
```

CLOBs and BLOBs are defined in `mle-js-plsqltypes`. Most commonly used types are provided in the global scope, rendering the import of `mle-js-plsqltypes` unnecessary.

The first step is to create a temporary, uncached LOB locator. Following the successful initialization of the LOB, it is opened for read and write operations. A string is written to the CLOB with an offset of `1`. Until this point, the LOB exists in memory. The call to `session.execute()` inserts the CLOB in the table. Calling the `close()` method closes the CLOB and frees the associated memory.

Reading LOBs

An example is used to show how to select a CLOB and then use the `fetchInfo` property to read the contents of the CLOB as a string.

Reading an LOB from the database is no different from reading other columns. Example 7-18 demonstrates how to fetch the row inserted by procedure `insert_clob`, defined in Example 7-17.

Example 7-18 Read an LOB

```
CREATE OR REPLACE FUNCTION read_clob(
    "p_id" NUMBER
) RETURN VARCHAR2
AS MLE LANGUAGE JAVASCRIPT
{{
const result = session.execute(
    `SELECT c
     FROM mle_lob_example
     WHERE id = :id`,
    {
        id:{
            type: oracledb.NUMBER,
            dir: oracledb.BIND_IN,
            val: p_id
```

```
            }
        },
        {
            fetchInfo:{
                "C": {type: oracledb.STRING}
            },
            outFormat: oracledb.OBJECT
        }
);
if (result.rows.length === 0){
    throw new Error(`No data found for ID ${id}`);
} else {
    for (let row of result.rows){
        return row.C;
    }
}
}};
/
```

The function `read_clob` receives an ID as a parameter. It is used in the select statement's `WHERE` clause as a bind variable to identify a row containing the CLOB. The `fetchInfo` property passed using `session.execute()` instructs the database to fetch the CLOB as a string.

API Differences Between node-oracledb and mle-js-oracledb

There are several differences between `node-oracledb` and `mle-js-oracledb`, including the methods for handling connection management and type mapping.

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about JavaScript built-in modules

Topics

- Synchronous API and Error Handling
- Connection Handling
- Transaction Management
- Type Mapping
- Unsupported Data Types
- Miscellaneous Features Not Available with the MLE JavaScript SQL Driver

Synchronous API and Error Handling

Compared to `node-oracledb`, the `mle-js-oracledb` driver operates in a synchronous mode, throwing exceptions as they happen. If an asynchronous behavior is desired, calls to `mle-js-oracledb` can be wrapped into async functions.

During synchronous operations, API calls block until either a result or an error are returned. Errors caused by SQL execution are reported as JavaScript exceptions, otherwise they return the same properties as the `node-oracledb Error` object.

The following methods neither return a promise nor do they take a callback parameter. They either return the result or throw an exception.

- `connection.execute`
- `connection.executeMany`
- `connection.getStatementInfo`
- `connection.getSodaDatabase`
- `connection.commit`
- `connection.rollback`
- `resultset.close`
- `resultset.getRow`
- `resultset.getRows`

The following method cannot be implemented in a synchronous way and is omitted in the MLE JavaScript driver.

- `connection.break`

`node-oracledb` provides a LOB (Large Object) class to provide streaming access to LOB types. The LOB class implements the asynchronous Node.js Stream API and cannot be supported in the synchronous MLE JavaScript environment. Large objects are supported using an alternative API in the MLE JavaScript driver. For these reasons, the following LOB-related functionality is not supported.

- `connection.createLob`
- `property oracledb.lobPrefetchSize`
- `constant oracledb.BLOB`
- `constant oracledb.CLOB`

`node-oracledb` also implements asynchronous streaming of query results, another feature that's based on the Node.js Stream API. A streaming API cannot be represented in a synchronous interface as used by the MLE JavaScript driver, therefore the following functionality is not available.

- `connection.queryStream()`
- `resultSet.toQueryStream()`

Connection Handling

The method of connection handling with the MLE JavaScript driver is described.
All SQL statements that are executed via the server-side MLE JavaScript driver are executed in the current session that is running the JavaScript program. SQL statements are executed with the privileges of the user on whose behalf JavaScript code is executed. As in the `node-oracledb` API, JavaScript code using the MLE JavaScript driver must acquire a Connection object to execute SQL statements. However, the only connection available is the implicit connection to the current database session.

JavaScript code must acquire a connection to the current session using the MLE-specific `oracledb.defaultConnection()` method. On each invocation, it returns a connection object that represents the session connection. Creation of connections with the `oracledb.createConnection` method of `node-oracledb` is not supported by the MLE JavaScript driver; neither is the creation of a connection pool supported. Connection objects are implicitly closed and so the call to `connection.close()` is not available with the MLE JavaScript driver.

There is also no statement cursor caching with the MLE JavaScript driver and therefore there is no `stmtCacheSize` property.

The Real Application Cluster (RAC) option offers additional features, designed to increase availability of applications. These include Fast Application Notification (FAN) and Runtime Load Balancing (RLB), neither of which are supported by the MLE JavaScript driver.

Transaction Management

With respect to transaction management, server-side MLE JavaScript code behaves exactly like PL/SQL procedures and functions.
A JavaScript program is executed in the current transaction context of the calling SQL or PL/SQL statement. An ongoing transaction can be controlled by executing `COMMIT`, `SAVEPOINT`, or `ROLLBACK` commands. Alternatively, the methods `connection.commit()` and `connection.rollback()` can be used.

MLE JavaScript SQL driver connections cannot be explicitly closed. Applications relying on `node-oracledb` behavior where closing a connection performs a rollback of the transaction will need adjusting. The MLE JavaScript SQL driver neither performs implicit commit nor rollback of transactions.

The `node-oracledb` driver features an auto-commit flag, defaulting to false. The MLE JavaScript SQL driver does not implement this feature. If specified, the `connection.execute()` function ignores the parameter.

Type Mapping

The MLE JavaScript driver adheres to the behavior of `node-oracledb` with respect to conversions between PL/SQL types and JavaScript types.
By default, PL/SQL types map to native JavaScript types (except for BLOBs and CLOBs). Values fetched from query results are implicitly converted. See MLE Type Conversions for more details about MLE type mappings.

As with `node-oracledb`, the conversion from non-character data types and vice versa is directly impacted by the NLS session parameters. The MLE runtime locale has no impact on these conversions.

To avoid loss of precision when converting between native JavaScript types and PL/SQL data types, the MLE JavaScript driver introduces new wrapper types.

- `oracledb.ORACLE_NUMBER`
- `oracledb.ORACLE_CLOB`
- `oracledb.ORACLE_BLOB`
- `oracledb.ORACLE_TIMESTAMP`
- `oracledb.ORACLE_TIMESTAMP_TZ`
- `oracledb.ORACLE_DATE`

- `oracledb.ORACLE_INTERVAL_YM`
- `oracledb.ORACLE_INTERVAL_DS`

As with `node-oracledb`, the default mapping to JavaScript types may be overridden on a case-by-case basis using the `fetchInfo` property on `connection.execute()`. Type constants like `oracledb.ORACLE_NUMBER` may be used to override the type mapping for a specific `NUMBER` column in order to avoid implicit conversion and loss of precision.

Additionally, the JavaScript MLE SQL driver provides a way to change the default mapping of PL/SQL types globally. If the `oracledb.fetchAsPlsqlWrapper` property contains the corresponding type constant, Oracle values are fetched as SQL wrapper types previously described. As with the existing property `oracledb.fetchAsString`, this behavior can be overridden using `fetchInfo` and `oracledb.DEFAULT`. Because MLE JavaScript does not support a `Buffer` class, and instead uses `Uint8Array`, property `oracledb.fetchAsBuffer` from `node-oracledb` does not exist in `mle-js-oracledb`, which instead uses `oracledb.fetchAsUint8Array`.

Changing the type mapping to fetch JavaScript SQL wrapper types by default accounts for the following scenarios:

- Oracle values are mainly moved between queries and DML statements, so that the type conversions between PL/SQL and JavaScript types are an unnecessary overhead.
- It is crucial to avoid data loss.

Example 7-19 Using JavaScript Native Data Types vs Using Wrapper Types

This example demonstrates the effect of using JavaScript native data types for calculations. It also compares the loss of precision using JavaScript native types versus using wrapper types.

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

/**
 *There is a potential loss of precision when using native
 *JavaScript types to perform certain calculations. This
 *is caused by the underlying implementation as a floating
 *point number
*/

export function precisionLoss(){

    let summand1 = session
        .execute(`SELECT 0.1 summand1`)
        .rows[0].SUMMAND1;

    let summand2 = session
        .execute(`SELECT 0.2 summand2`)
        .rows[0].SUMMAND2;

    const result = summand1 + summand2;

    console.log(`precisionLoss() result: ${result}`);
}

/**
 *Use an Oracle data type to preserve precision. The above
```

```
 *example can be rewritten using the OracleNumber type as
 *follows
*/
export function preservePrecision(){

    //instruct the JavaScript SQL driver to return results as
    //Oracle Number. This could have been done for individual
    //statements using the fetchInfo property - the global
    //change applies to this and all future calls
    oracledb.fetchAsPlsqlWrapper = [oracledb.NUMBER];
    let summand1 = session
        .execute(`SELECT 0.1 S1`)
        .rows[0].S1;

    let summand2 = session
        .execute(`SELECT 0.2 S2`)
        .rows[0].S2;

    const result = summand1 + summand2;

    console.log(`preservePrecision() result: ${result}`);
}
/
```

When executing the above functions, the difference in precision becomes immediately obvious.

```
precisionLoss() result: 0.30000000000000004
preservePrecsion() result: .3
```

Rather than setting the global `oracledb.fetchAsPlsqlWrapper` property, it is possible to override the setting per invocation of `connection.execute()`. Example 7-20 shows how `precisionPreservedGlobal()` can be rewritten by setting precision inline.

For information about functions available for use with type `OracleNumber`, see Server-Side JavaScript API Documentation.

Example 7-20 Overriding the Global oracledb.fetchAsPlsqlWrapper Property

This example extends Example 7-19 by showing how `precisionPreservedGlobal()` can be rewritten by preserving precision inline. It demonstrates that rather than setting the global `oracledb.fetchAsPlsqlWrapper` property, it is possible to override the setting per invocation of `connection.execute()`.

```
CREATE OR REPLACE PROCEDURE fetch_info_example
AS MLE LANGUAGE JAVASCRIPT
{{
    let summand1 = session
        .execute(
            `SELECT 0.1 S1`,
            [],
            {
                fetchInfo:{
                    S1:{type: oracledb.ORACLE_NUMBER}
                }
            }
```

```
        )
        .rows[0].S1;

    let summand2 = session
        .execute(
            `SELECT 0.2 S2`,
            [],
            {
                fetchInfo:{
                    S2:{type: oracledb.ORACLE_NUMBER}
                }
            }
        )
        .rows[0].S2;

    const result = summand1 + summand2;

    console.log(`
    preservePrecision():
    summand1: ${summand1}
    summand2: ${summand2}
    result: ${result}
    `);
}};
/
```

Result:

```
preservePrecision():
summand1: .1
summand2: .2
result: .3
```

Unsupported Data Types

The MLE JavaScript driver does not currently support these data types:

- `LONG`
- `LONG RAW`
- `XMLType`
- `BFILE`
- `REF CURSOR`

Miscellaneous Features Not Available with the MLE JavaScript SQL Driver

Differences between what features are available with the MLE JavaScript driver and with `node-oracledb` are described.
Error handling in the MLE JavaScript driver relies on the JavaScript exception framework rather than using a callback/promise as `node-oracledb` does. The error thrown by the MLE JavaScript SQL driver is identical to the `Error` object available with `node-oracledb`.

Several additional client-side features available in `node-oracledb` are not supported by the server-side MLE environment. The MLE JavaScript driver omits the API for these features.

The following features are currently unavailable:

- Continuous Query Notification (CQN)
- Advanced Queuing is not supported natively, the PL/SQL API can be used as a workaround
- `Connection.subscribe()`
- `Connection.unsubscribe()`
- All Continuous Query Notification constants in the `oracledb` class
- All Subscription constants in the `oracledb` class

Introduction to the PL/SQL Foreign Function Interface

The Foreign Function Interface (FFI) is designed to provide straightforward access to PL/SQL packages in a familiar, JavaScript-like fashion.

Using the `mle-js-plsql-ffi` API, wrappers are created around PL/SQL packages and procedures so that in subsequent calls, you can interact with them as if they were JavaScript objects and functions. This approach can be used in certain cases as an alternative to using the MLE JavaScript SQL driver.

A lot of database functionality is available in the form of PL/SQL packages; either built-in, those installed by frameworks such as APEX, or user-defined PL/SQL code. The Foreign Function Interface (FFI) allows you to access PL/SQL functionality in packages and procedures directly from JavaScript code without executing SQL statements, providing a seamless integration of existing PL/SQL functionality with server-side JavaScript applications. For example, database procedures can be invoked as JavaScript functions, passing JavaScript values as function arguments.

Consider the following JavaScript snippet that uses `session.execute` to employ the `DBMS_RANDOM` package inside an anonymous PL/SQL block:

```
CREATE OR REPLACE FUNCTION get_random_number(
    p_lower_bound NUMBER,
    p_upper_bound NUMBER
) RETURN NUMBER
AS MLE LANGUAGE JAVASCRIPT
{{
    const result = session.execute(
        'BEGIN :randomNum := DBMS_RANDOM.VALUE(:low, :high); END;',
        {
            randomNum: {
                type: oracledb.NUMBER,
                dir: oracledb.BIND_OUT
            }, low: {
                type: oracledb.NUMBER,
                dir: oracledb.BIND_IN,
                val: P_LOWER_BOUND
            }, high: {
                type: oracledb.NUMBER,
                dir: oracledb.BIND_IN,
                val: P_UPPER_BOUND
```

```
            }
        }
    );

    return result.outBinds.randomNum;
}};
/

SELECT get_random_number(1,100);
```

Using FFI, you can cut down on the boilerplate code needed to implement the previous example. The following snippet achieves the same functionality as the previous one in a more concise way:

```
CREATE OR REPLACE FUNCTION get_random_number(
    p_lower_bound NUMBER,
    p_upper_bound NUMBER
) RETURN NUMBER
AS MLE LANGUAGE JAVASCRIPT
{{
    const { resolvePackage } = await import ('mle-js-plsql-ffi');

    const dbmsRandom = resolvePackage('dbms_random');

    return dbmsRandom.value(P_LOWER_BOUND, P_UPPER_BOUND);
}};
/

SELECT get_random_number(1,100);
```

- Object Resolution Using FFI
  A set of functions is available with the `mle-js-plsql-ffi` API, each returning a JavaScript object that represents its database counterpart.
- Provide Arguments to a Subprogram Using FFI
  Use the `arg` and `argOf` functions to handle `IN OUT` and `OUT` parameters with the Foreign Function Interface (FFI).

> **See Also:**
>
> Server-Side JavaScript API Documentation for more information about the `mle-js-plsql-ffi` API

Object Resolution Using FFI

A set of functions is available with the `mle-js-plsql-ffi` API, each returning a JavaScript object that represents its database counterpart.

The following functions are available to resolve packages and top-level functions and procedures:

- `resolvePackage('<pkg_name>')`

- `resolveProcedure('<proc_name>')`
- `resolveFunction('<func_name>')`

If the object you want to resolve is in your own schema or has a public synonym, qualifying the object name with the owning schema is optional. If the object is in a different schema, you must have necessary permissions to access the object and must qualify its name with the owning schema. As with the MLE JavaScript SQL driver, all operations are performed in your own security context.

> **Note:**
>
> If the named database object does not exist or you do not have access to it, a `RangeError` is raised. If the given name resolves to a database object that is not the correct type, a `TypeError` is raised. Database links are not supported. Attempting to resolve a name with a database link results in an `Error`.

> **Note:**
>
> The provided FFI functions follow the same case-sensitivity rules as PL/SQL, meaning names are auto-capitalized by default. For quoted identifiers, you must use JavaScript dictionary notation with a combination of double and single quotes to indicate case-sensitivity:
>
> ```
> // call a procedure with case-sensitive name
> myPkg['"MyProc"']();
>
> // read a global variable with a case-sensitive name
> console.log(myPkg['"MyVar"']);
> ```

Once a database object has been resolved, you can perform the following operations on the resulting object:

- **Procedure:** Execute
- **Function:** Execute
- **Package:**
  - Execute procedure
  - Execute function
  - Read and write public package variables
  - Read constants

With `resolvePackage`, variables, constants, procedures, and functions can be accessed directly through property reads of the resulting object. If the package does not have the member provided in the property read, a `Reference` error is thrown. When the accessed member is a PL/SQL function or procedure, the JavaScript object returns the same type of

callable entity that is resolved for top level functions and procedures. Consider the following snippets for examples of the syntax:

```
// resolve a package
const myPkg = resolvePackage('my_package');

// call a procedure and function in the package
myPkg.my_proc();
let result = myPkg.my_func();

// read a global variable and constant in the package
console.log(myPkg.my_var);
console.log(myPkg.my_const);

// write a global variable in the package
myPkg.my_var = 42;
```

For package variables and constants, only non-named types are supported. The following types are not supported: PL/SQL record types, nested table types, associative arrays, vector types, and ADTs.

When resolving a procedure or function, you receive a callable object. With functions, the `overrideReturnType` instance method can optionally be used to specify the return type and change other metadata. Consider the following example that uses `overrideReturnType` to increase the `maxSize` attribute:

1. Start by creating a function that returns a string:

   ```
   CREATE OR REPLACE FUNCTION ret_string(
       MULTIPLIER NUMBER
   ) RETURN VARCHAR2 AS
   BEGIN
       return rpad('this string might be too long for the defaults ',
   MULTIPLIER, 'x');
   END;
   /
   ```

2. Create another function, `ret_string_ffi`, that uses FFI to resolve the function `ret_string`:

   ```
   CREATE OR REPLACE FUNCTION ret_string_ffi(
       MULTIPLIER NUMBER
   ) RETURN VARCHAR2
   AS MLE LANGUAGE JAVASCRIPT
   {{
       const retStrFunc = plsffi.resolveFunction('ret_string');
       return retStrFunc(MULTIPLIER);
   }};
   /
   ```

3. The `ret_string_ffi` function will work as long as the multiplier value is small enough, as in the following:

   ```
   SELECT ret_string_ffi(50);
   ```

Result:

```
RET_STRING_FFI(50)
--------------------------------------------------------------------------------
-----
this string might be too long for the defaults xxx
```

4. With a larger multiplier value, the result can exceed the default buffer length of 200 bytes and raise an error:

```
SELECT ret_string_ffi(900);
```

Result:

```
SELECT ret_string_ffi(900)
                      *
ERROR at line 1:
ORA-04161: Error: Exception during subprogram execution (6502): ORA-06502:
PL/SQL: value or conversion error: character string buffer too small
ORA-04171: at :=> (<inline-src-js>:3:12)
```

5. You can solve this problem by using the `overrideReturnType` instance method to increase the `maxSize` attribute of the returned message:

```
CREATE OR REPLACE FUNCTION ret_str_ffi_override(
    MULTIPLIER NUMBER
) RETURN VARCHAR2
AS MLE LANGUAGE JAVASCRIPT
{{
    const retStrFunc = plsffi.resolveFunction('ret_string');

    // overrideReturnType accepts either an oracledb type constant
    // such as oracledb.NUMBER, or a string containing the name of a
    // user defined database type. If more information is needed, as
    // in this example, a parameter of type ReturnInfo can be provided
    retStrFunc.overrideReturnType({
        maxSize: 1000
    });
    return retStrFunc(MULTIPLIER);
}};
/
```

6. Using the new `ret_str_ffi_override` function, a call with a larger multiplier will now work:

```
SELECT ret_str_ffi_override(900);
```

Provide Arguments to a Subprogram Using FFI

Use the `arg` and `argOf` functions to handle `IN OUT` and `OUT` parameters with the Foreign Function Interface (FFI).

JavaScript and PL/SQL handle parameters differently. For instance, JavaScript doesn't allow for named parameters in the same way that PL/SQL does. Neither does JavaScript have an equivalent for `OUT` and `IN OUT` parameters, nor is there an option for overloading functions.

Last, but not least, JavaScript types are different from the database's built-in type system. To be able to call PL/SQL from JavaScript, the FFI must accommodate these differences.

For more information about PL/SQL subprogram parameters, see *Oracle Database PL/SQL Language Reference*.

The following procedure represents a case where:

- multiple parameters are defined.
- parameters provide a mix of `IN`, `OUT`, and `IN OUT` modes.
- the default `maxSize` for a `VARCHAR2 OUT` variable is insufficient

```
CREATE OR REPLACE PROCEDURE my_proc_w_args(
    p_arg1      IN NUMBER,
    p_arg2      IN NUMBER,
    p_arg3      IN OUT JSON,
    p_arg4      OUT TIMESTAMP,
    p_arg5      OUT VARCHAR2
) AS
BEGIN

  SELECT
    JSON_TRANSFORM(p_arg3,
      SET '$.lastUpdate' = systimestamp,
      SET '$.value' = p_arg1 + p_arg2
    )
    into p_arg3;

  p_arg4 := systimestamp;

  -- the length of the string will exceed the default
  -- length of 200 characters for the out bind, mandating
  -- the use of maxSize in args().
  p_arg5 := rpad('x', 255, 'x');

END;
/
```

Parameters passed using the `IN` mode do not require any special treatment. The FFI provides the `arg()` and `argOf()` functions to handle `OUT` and `IN OUT` parameters, respectively. Remember that all parameters provided using the FFI are essentially bind parameters and thus their behavior can be influenced using the same `dir`, `val`, `type`, and `maxSize` properties you use if you call PL/SQL directly using `session.execute()`.

The `arg` function generates an object that represents an argument. It optionally accepts the same object as the MLE JavaScript SQL driver, including any combination of the `dir`, `val`, `type`, and `maxSize` properties.

The `argOf` function generates an object that represents an argument of the given value.

Parameters can be passed in two different ways:

- As a list of positional arguments.
- Using an object to provide the arguments, simulating named parameters.

Based on the function created in the preceding example, `my_proc_w_args`, you can invoke the function with the FFI using *positional arguments* as follows:

```
CREATE OR REPLACE PROCEDURE my_proc_w_args_positional(
    "arg1" NUMBER,
    "arg2" NUMBER
) AS MLE LANGUAGE JAVASCRIPT
{{
    const myProc = plsffi.resolveProcedure('my_proc_w_args');

    // arg3 is an IN OUT parameter of type JSON. my_proc_with_args
    // will modify it in place and return it to the caller
    const arg3 = plsffi.argOf({id: 10, value: 100});

    // arg4 is a pure OUT parameter
    const arg4 = plsffi.arg();

    // arg5 represents an OUT parameter as well but due to the
    // length of the return string, it must be provided with additional
    // metadata
    const arg5 = plsffi.arg({
        maxSize: 1024
    });

    myProc(arg1, arg2, arg3, arg4, arg5);

    console.log(`the updated JSON looks like this: $
{JSON.stringify(arg3.val)}`);
    console.log(`the calculation happened at ${arg4.val}`);
    console.log(`the length of the string returned is ${arg5.val.length}
characters`);
}};
/
```

The second option is to use *named arguments*, provided as a single, plain JavaScript object. The FFI API then maps each property to the argument that matches the name of the property.

```
CREATE OR REPLACE PROCEDURE my_proc_w_args_named(
    "arg1" NUMBER,
    "arg2" NUMBER
) AS MLE LANGUAGE JAVASCRIPT
{{
    const myProc = plsffi.resolveProcedure('my_proc_w_args');

    // arg3 is an IN OUT parameter of type JSON. my_proc_with_args
    // will modify it in place and return it to the caller
    const arg3 = plsffi.argOf({id: 10, value: 100});

    // arg4 is a pure OUT parameter
    const arg4 = plsffi.arg();

    // arg5 represents an OUT parameter as well but due to the
    // length of the return string must be provided with additional
    // metadata
    const arg5 = plsffi.arg({
```

```
        maxSize: 1024
    });

    myProc({
        p_arg1: arg1,
        p_arg2: arg2,
        p_arg3: arg3,
        p_arg4: arg4,
        p_arg5: arg5
    });

    console.log(`the updated JSON looks like this: $
{JSON.stringify(arg3.val)}`);
    console.log(`the calculation happened at ${arg4.val}`);
    console.log(`the length of the string returned is ${arg5.val.length}
characters`);
}};
/
```

Note the edge case where you have a PL/SQL subprogram that has a single argument that is represented in JavaScript as an `object`. Intuitively, you may want to pass it as a single positional argument, however, in that case, the FFI will interpret it as a *named arguments* object.

There are two ways around this exception:

- You can wrap your argument in an object as if you were calling the subprogram with named arguments.
- You can wrap your argument with `plsffi.argOf()` and the FFI will recognize it as a single positional argument.

Consider the following example that demonstrates these options:

```
-- PL/SQL subprogram we want to call
CREATE OR REPLACE PROCEDURE my_proc(my_arg JSON) AS
BEGIN
    -- Process my_arg
END;

-- JavaScript function that calls my_proc
CREATE OR REPLACE PROCEDURE my_javascript_proc
AS MLE LANGUAGE JAVASCRIPT
{{
    const myProc = plsffi.resolveProcedure('my_proc');
    const myArg = { prop1: 10, prop2: 'foo' };

    // Catch the exception that will happen if the FFI tries
    // to interpret this as a call with named arguments
    try {
        myProc(myArg);
    } catch (err) {
        console.log(`if uncaught, this would have been a ${err}`);
    }

    // Option 1: Make it into a real named argument call.
    myProc({ my_arg: myArg });
```

```
    // Option 2: Wrap with argOf() to let the FFI know that it's a
    // positional argument list call.
    myProc(plsffi.argOf(myArg));
}};
```

PL/SQL allows developers to overload signatures of functions and procedures that are defined in PL/SQL packages. The FFI does not perform overload selection, however, it still needs to decide what PL/SQL type to use for binding each argument. Unfortunately, it cannot make this decision on its own in all cases. In particular, in the following instances:

- No JavaScript value was given for an argument that is needed to determine the correct signature to call. Without a value, the FFI has no way of knowing the set of matching PL/SQL types.
- When one JavaScript type is viable for multiple PL/SQL types.

Keep in mind that FFI uses SQL driver constants to represent standard types and strings (containing the type name) for user defined types. SQL driver constants come in two flavors:

- Constants that start with `DB_TYPE_*` control how the JavaScript value is converted to a PL/SQL value.
- All others are used to control how the returned PL/SQL value is converted to a JavaScript value.

If you are specifying the type of your argument in order to help with type resolution, it is best to use one of the `DB_TYPE_*` constants.

Consider the following PL/SQL package:

```
CREATE OR REPLACE package overload_pkg AS

    FUNCTION my_func(
        p_arg1 IN BINARY_FLOAT
    ) RETURN VARCHAR2;

    FUNCTION my_func(
        p_arg1 IN INTEGER
    ) RETURN VARCHAR2;
END;
/

CREATE OR REPLACE PACKAGE BODY overload_pkg AS

    FUNCTION my_func(
        p_arg1 IN BINARY_FLOAT
    ) RETURN VARCHAR2 AS
    BEGIN
        RETURN 'binary_float';
    END;

    FUNCTION my_func(
        p_arg1 IN INTEGER
    ) RETURN VARCHAR2 AS
    BEGIN
        RETURN 'integer';
    END;
```

```
END;
/
```

As you can see, `my_proc` is overloaded, accepting both a `BINARY_FLOAT` as well as an `INTEGER`. In JavaScript, both of these types are represented as the number data type and as such, multiple possible overloads are valid. If the FFI API cannot select the correct resolution, it is possible to force a particular overloaded PL/SQL function by providing the PL/SQL type.

```
CREATE OR REPLACE PROCEDURE force_overload
AS MLE LANGUAGE JAVASCRIPT
{{
    const myPkg = plsffi.resolvePackage('overload_pkg');

    let result = 'not yet called';

    // Catch error ORA-04161: Error: Exception during subprogram execution
    // (4161): Multiple subprograms match the provided signature
    try {
        result = myPkg.my_func(42);
    } catch (err) {
        console.log(`if uncaught, this would have been a ${err}`);
    }

    // Solution: use argOf to make this work
    result = myPkg.my_func(plsffi.argOf(42, {type:
oracledb.DB_TYPE_BINARY_FLOAT}))
    console.log(`and the result is: ${result}`);
}};
/
```

An error can also occur if the type is user-defined. For example, all JavaScript objects are considered viable for all PL/SQL records. In this case, it is enough to provide the name of the desired type.

8

Working with SODA Collections in MLE JavaScript Code

Simple Oracle Document Access (SODA) is a set of NoSQL-style APIs that let you create and store collections of documents (in particular JSON) in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the documents are stored in the database.

SODA APIs exist for different programming languages and include support for MLE JavaScript. SODA APIs are *document-centric*. You can use any SODA implementation to perform create, read, update, and delete (CRUD) operations on documents of nearly any kind (including video, image, sound, and other binary content). You can also use any SODA implementation to query the content of JavaScript Object Notation (JSON) documents using pattern-matching: query-by-example (QBE). CRUD operations can be driven by document keys or by QBEs.

This chapter covers JavaScript in the database, based on Multilingual Engine (MLE) as opposed to the client-side `node-oracledb` driver. Whenever JavaScript is mentioned in this chapter it implicitly refers to MLE JavaScript.

> **Note:**
>
> In order to use the MLE SODA API, the `COMPATIBLE` initialization parameter must be set to `23.0.0`.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for a complete introduction to SODA

Topics

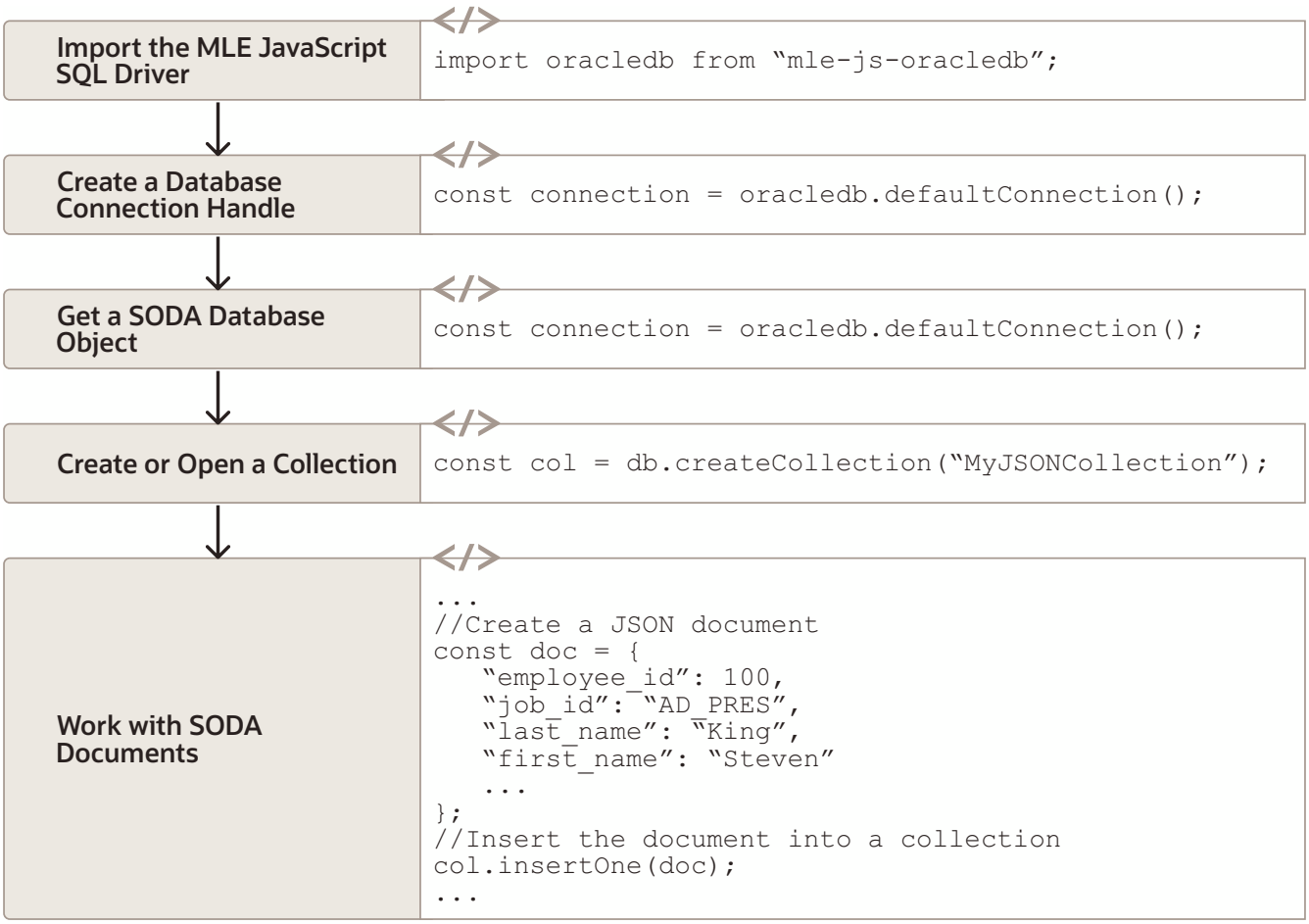
High-Level Introduction to Working with SODA for In-Database JavaScript

The SODA API is part of the MLE JavaScript SQL driver. Interaction with collections and documents requires you to establish a connection with the database first, before a SODA database object can be obtained.

The SODA database is the top-level abstraction object when working with the SODA API.

Figure 8-1 demonstrates the standard control flow.

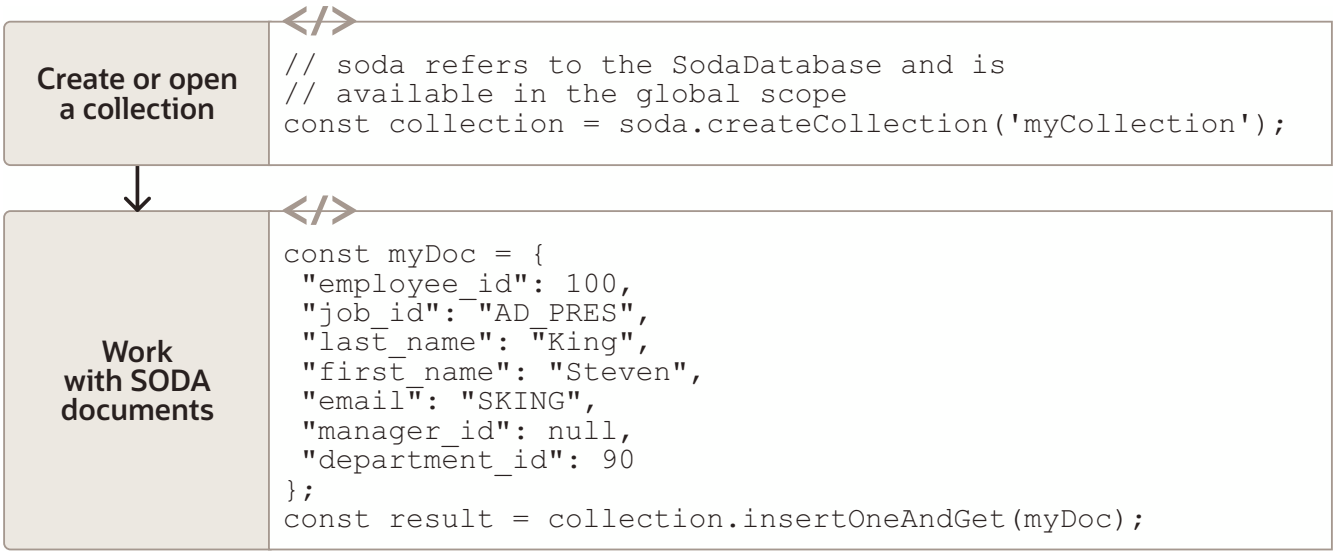
Figure 8-1 SODA for In-Database JavaScript Basic Workflow



Applications that aren't ported from client-side Node.js or Deno can benefit from coding aids available in the MLE JavaScript SQL driver, such as a number of frequently used variables that are available in the global scope. For a complete list of available global variables and types, see Working with the MLE JavaScript Driver.

For SODA applications the most important global variable is the `soda` object, which represents the `SodaDatabase` object. The availability of the `soda` object in the global scope reduces the need for writing boilerplate code. In this case the workflow can be simplified, as in Figure 8-2.

Figure 8-2 SODA for In-Database JavaScript Simplified Workflow



> **Note:**
>
> If you are running your JavaScript code in a restricted execution context, you cannot use the SODA API. For more information about restricted execution contexts, see About Restricted Execution Contexts.

SODA Objects

Objects used with the SODA API.

The following objects are at the core of the SODA API:

- **`SodaDatabase`:** The top-level object for SODA operations. This is acquired from an Oracle Database connection or directly available from the global scope as the `soda` object. A SODA database is an abstraction, allowing access to SODA collections in that SODA database, which then allow access to documents in those collections. A SODA database is analogous to an Oracle Database user or schema. A collection is analogous to a table. A document is analogous to a table row with one column for a unique document key, a column for the document content, and other columns for various document attributes. With the MLE JavaScript SQL driver, the `soda` object is available as a global variable, which represents the `SodaDatabase` object and reduces the need for writing boilerplate code.
- **`SodaCollection`:** Represents a collection of SODA documents. By default, collections allow JSON documents to be stored, and they add a default set of metadata to each document. This is recommended for most users. However, optional metadata can set various details about a collection, such as its database storage, whether it should track version and time stamp document components, how such components are generated, and what document types are supported. Most users do not need to provide custom metadata.
- **`SodaDocument`:** Represents a document. Typically, the document content will be JSON. The document has properties including the content, a key, timestamps, and the media type. By default, document keys are automatically generated.

When working with collections and documents stored therein, you will make use of the following objects:

- **`SodaDocumentCursor`:** A cursor object representing the result of the `getCursor()` method from a `find()` operation. It can be iterated over to access each `SodaDocument`.
- **`SodaOperation`:** An internal object used with `find()` to perform read and write operations on documents. Chained methods set properties on a `SodaOperation` object which is then used by a terminal method to find, count, replace, or remove documents. This is an internal object that should not be directly accessed.

> **See Also:**
>
> Server-Side JavaScript API Documentation for information about using SODA objects with `mle-js-oracledb`

Using SODA for In-Database JavaScript

How to access SODA for In-Database JavaScript is described, as well as how to use it to perform create, read (retrieve), update, and delete (CRUD) operations on collections.

This section describes SODA for MLE JavaScript. Code snippets in this section are sometimes abridged for readability. Care has been taken to ensure that JavaScript functions are listed in their entirety, but they aren't runnable on their own. Embedding the function definition into a JavaScript module and importing the MLE JavaScript SQL driver will convert these code examples to valid JavaScript code for Oracle Database 23ai.

Topics

- Getting Started with SODA for In-Database JavaScript
  How to access SODA for In-Database JavaScript is described, as well as how to use it to create a database collection, insert a document into a collection, and retrieve a document from a collection.
- Creating a Document Collection with SODA for In-Database JavaScript
  How to use SODA for In-Database JavaScript to create a new document collection is explained.
- Opening an Existing Document Collection with SODA for In-Database JavaScript
  You can use the method `SodaDatabase.openCollection()` to open an existing document collection or to test whether a given name names an existing collection.
- Checking Whether a Given Collection Exists with SODA for In-Database JavaScript
  You can use `SodaDatabase.openCollection()` to check for the existence of a given collection. It returns `null` if the collection argument does not name an existing collection; otherwise, it opens the collection having that name.
- Discovering Existing Collections with SODA for In-Database JavaScript
  You can use `SodaDatabase.getCollectionNames()` to fetch the names of all existing collections for a given `SodaDatabase` object.
- Dropping a Document Collection with SODA for In-Database JavaScript
  You use `SodaCollection.drop()` to drop an existing collection.
- Creating Documents with SODA for In-Database JavaScript
  Creation of documents by SODA for In-Database JavaScript is described.

- Inserting Documents into Collections with SODA for In-Database JavaScript
  `SodaCollection.insertOne()` or a related call such as `sodaCollection.insertOneAndGet()` offers convenient ways to add documents to a collection. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key, which is not recommended for must users.
- Saving Documents into Collections with SODA for In-Database JavaScript
  You use `SodaCollection.save()` and `saveAndGet()` to save documents into collections.
- SODA for In-Database JavaScript Read and Write Operations
  The primary way you specify read and write operations (other than insert and save) is to use methods provided by the `SodaOperation` class. You can chain together `SodaOperation` methods to specify read or write operations against a collection.
- Finding Documents in Collections with SODA for In-Database JavaScript
  To find documents in a collection, you invoke `SodaCollection.find()`. It creates and returns a `SodaOperation` object which is used via method chaining with nonterminal and terminal methods.
- Replacing Documents in a Collection with SODA for In-Database JavaScript
  To replace the content of one document in a collection with the content of another, you start by looking up the document to be modified using its key. Because `SodaOperation.key()` is a nonterminal operation, the easiest way to replace the contents is to chain `SodaOperation.key()` to `SodaOperation.replaceOne()` or `SodaOperation.replaceOneAndGet()`.
- Removing Documents from a Collection with SODA for In-Database JavaScript
  Removing documents from a collection is similar to replacing. The first step is to perform a lookup operation, usually based on the document's key or by using a search expression in `SodaOperation.filter()`. The call to `SodaOperation.remove()` is a terminal operation, in other words the last operation in the chain.
- Indexing the Documents in a Collection with SODA for In-Database JavaScript
  Indexes can speed up data access, regardless of whether you use the NoSQL style SODA API or a relational approach. You index documents in a SODA collection using `SodaCollection.createIndex()`. Its `IndexSpec` parameter is a textual JSON index specification.
- Getting a Data Guide for a Collection with SODA for In-Database JavaScript
  A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents. They provide great insights into JSON documents and are invaluable for getting an overview of a data set.
- Handling Transactions with SODA for In-Database JavaScript
  Unlike the client-side JavaScript SQL driver, the MLE JavaScript SQL driver does not provide an `autoCommit` feature. You need to commit or roll your transactions back, either in the PL/SQL layer in case of module calls, or directly in the JavaScript code by calling `connection.commit()` or `connection.rollback()`.
- Creating Call Specifications Involving the SODA API
  Earlier in this chapter, in the section *Getting Started with SODA for In-Database JavaScript*, an example showing how to invoke the MLE SODA API using an inline call specification is included. The following short example demonstrates how to use SODA in MLE modules.

Getting Started with SODA for In-Database JavaScript

How to access SODA for In-Database JavaScript is described, as well as how to use it to create a database collection, insert a document into a collection, and retrieve a document from a collection.

Before you can get started working with SODA for MLE JavaScript, the account used for storing collections (in this case, `emily`) must be granted the `SODA_APP` roles, either directly or using the `DB_DEVELOPER_ROLE`:

```
grant soda_app to emily
```

Accessing SODA functionality requires the use of the MLE JavaScript SQL driver. Because the database session exists by the time the code is invoked, no additional connection handling is necessary. Example 8-1 demonstrates how to:

- Create a SODA collection,
- Insert a JSON document into it, and
- Iterate over all SODA Documents in the collection, printing their contents on screen

Each concept presented by Example 8-1 - creating collections, adding and modifying documents, and dropping collections - is addressed in more detail later in this chapter.

Example 8-1 SODA with MLE JavaScript General Workflow

This example demonstrates the general workflow using SODA collections with MLE JavaScript. Instead of using an MLE module, the example simplifies the process by implementing an inline call specification.

```
CREATE OR REPLACE PROCEDURE intro_soda(
    "dropCollection" BOOLEAN
) AUTHID CURRENT_USER
AS MLE LANGUAGE JAVASCRIPT
{{

  // use the soda object, available in the global scope instead of importing
  // the mle-js-oracledb driver, getting the default connection and extracting
  // the SodaDatabase from it
  const col = soda.createCollection("MyCollection");

  // create a JSON document (based on the HR.EMPLOYEES table for the employee
with id 100)
  const doc = {
    "_id" : 100,
    "job_id" : "AD_PRES",
    "last_name" : "King",
    "first_name" : "Steven",
    "email" : "SKING",
    "manager_id" : null,
    "department_id" : 90
  };

  // insert the document into collection
  col.insertOne(doc);
```

```
  // find all documents in the collection and print them on screen
  // use a cursor to iterate over all documents in the collection
  const c = col.find()
            .getCursor();

  let resultDoc;

  while (resultDoc = c.getNext()){
    const content = resultDoc.getContent();
    console.log(`
      ----------------------------------------
      key:           ${resultDoc.key}
      content (select fields):
      - _id:         ${content._id}
      - job_id:      ${content.job_id}
      - name:        ${content.first_name} ${content.last_name}
      version:       ${resultDoc.version}
      media type:    ${resultDoc.mediaType}`
    );
  }

  // it is very important to close the SODADocumentCursor to free resources
  c.close();

  // optionally drop the collection
  if (dropCollection){
    // there is no auto-commit, the outstanding transaction must be
    // finished before the collection can be dropped
    session.commit();
    col.drop();
  }
}};
/
```

You can try the code by executing the procedure using your favorite IDE. Here is an example of the results of calling the `intro_soda` procedure:

```
BEGIN
  intro_soda(true);
END;
/
```

Result:

```
----------------------------------------
key:           03C202
content (select fields):
- _id:         100
- job_id:      AD_PRES
- name:        Steven King
version:       17EF0F3C102653DDE063DA464664399C
media type:    application/json
```

```
PL/SQL procedure successfully completed.
```

Creating a Document Collection with SODA for In-Database JavaScript

How to use SODA for In-Database JavaScript to create a new document collection is explained.

Collections allow you to logically group documents. Before a collection can be created or accessed, a few more steps must be completed unless you make use of the global `soda` object. Begin by creating a connection object. The connection object is the starting point for all SODA interactions in the MLE JavaScript module:

```
// get a connection handle to the database session
const connection = oracledb.defaultConnection();
```

Once the connection is obtained, you can use it to call `Connection.getSodaDatabase()`, a prerequisite for creating the collection:

```
// get a SODA database
const db = connection.getSodaDatabase();
```

With the SODA database available, the final step is to create the collection. Note that collection names are case-sensitive:

```
// Create a collection with the name "MyCollection".
// This creates a database table, also named "MyCollection",
// to store the collection. If a collection with the same name
// exists, it will be opened
const col = db.createCollection("MyCollection");
```

The preceding statement creates a collection that, by default, allows JSON documents to be stored. If the collection name passed to `SodaDatabase.createCollection()` is that of an existing collection, it will simply be opened. You can alternatively open a known, existing collection using `SodaDatabase.openCollection()`.

Unless custom metadata is provided to `SodaDatabase.createCollection()` (which is not recommended), *default collection metadata* will be supplied. The default metadata has the following characteristics:

- Each document in the collection has these components:
  - Key
  - Content
  - Version
- The collection can store only JSON documents.
- Document keys and version information are generated automatically.

Optional collection metadata can be provided to the call to `createCollection()`, however, the default collection configuration is recommended in most cases.

If a collection with the same name already exists, it is simply opened and its object is returned. If custom metadata is passed to the method and does not match that of the existing collection,

the collection is not opened and an error is raised. To match, all metadata fields must have the same values.

> **See Also:**
>
> *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for more details about collection metadata, including custom metadata.

Opening an Existing Document Collection with SODA for In-Database JavaScript

You can use the method `SodaDatabase.openCollection()` to open an existing document collection or to test whether a given name names an existing collection.

Example 8-2 Opening an Existing Document Collection

This example opens the collection named `collectionName`. It is very important to check that the collection object returned by `SodaDatabase.openCollection()` is not `null`. Rather than throwing an error, the method will return a `null` value should the requested collection not exist.

```
export function openCollection(collectionName) {

    // perform a lookup. If a connection cannot be found by that
    // name no exception nor error are thrown, but the resulting
    // collection object will be null
    const col = soda.openCollection(collectionName);
    if (col === null) {
        throw new Error(`No such collection ${collectionName}`);
    }

    // do something with the collection
}
```

Checking Whether a Given Collection Exists with SODA for In-Database JavaScript

You can use `SodaDatabase.openCollection()` to check for the existence of a given collection. It returns `null` if the collection argument does not name an existing collection; otherwise, it opens the collection having that name.

In Example 8-2, if `collectionName` does not name an existing collection then `col` is assigned the value `null`.

Discovering Existing Collections with SODA for In-Database JavaScript

You can use `SodaDatabase.getCollectionNames()` to fetch the names of all existing collections for a given `SodaDatabase` object.

If the number of collections is very large, you can limit the number of names returned. Additionally, the lookup can be limited to collections starting with a user-defined string as demonstrated by Example 8-4.

Example 8-3 Fetching All Existing Collection Names

This example prints the names of all existing collections using the method `getCollectionNames()`.

```
export function printCollectionNames(){
  // loop over all collections in the current user's schema
  const allCollections = soda.getCollectionNames();
  for (const col of allCollections){
    console.log(`- ${col}`);
  }
}
```

Example 8-4 Filtering the List of Returned Collections

This example limits the results of `getCollectionNames()` by only printing the names of collections that begin with a user-defined string, `startWith`.

```
export function printSomeCollectionNames(numHits, startWith) {

  // loop over all collections in the current schema, limited
  // to those that start with a specific character sequence and
  // a maximum number of hits returned
  const allCollections = soda.getCollectionNames(
    {
      limit: numHits,
      startsWith: startWith
    }
  );
  for (const col of allCollections){
    console.log(`-${col}`);
  }
}
```

Dropping a Document Collection with SODA for In-Database JavaScript

You use `SodaCollection.drop()` to drop an existing collection.

> **Caution:**
>
> Do *not* use SQL to drop the database *table* that underlies a collection. Dropping a *collection* involves more than just dropping its database table. In addition to the documents that are stored in its table, a collection has *metadata*, which is also persisted in Oracle Database. Dropping the table underlying a collection does *not* also drop the collection metadata.

> **Note:**
>
> Day-to-day use of a typical application that makes use of SODA does not require that you drop and re-create collections. But if you need to do that for any reason then this guideline applies.
>
> Do *not* drop a collection and then re-create it with *different metadata* if there is any application running that uses the collection in any way. Shut down any such applications before re-creating the collection, so that all live SODA objects are released.
>
> There is no problem just dropping a collection. Any read or write operation on a dropped collection raises an error. And there is no problem dropping a collection and then re-creating it with the same metadata. But if you re-create a collection with different metadata, and if there are any live applications using SODA objects, then there is a risk that a stale collection is accessed, and *no error is raised* in this case.
>
> In SODA implementations that allow collection metadata caching, such as SODA for Java, this risk is increased if such caching is enabled. In that case, a (shared or local) cache can return an entry for a stale collection object even if the collection has been dropped.

> **Note:**
>
> Commit all writes to a collection before using `SodaCollection.drop()`. For the method to succeed, all uncommitted writes to the collection must first be committed. Otherwise, an exception is raised.

Example 8-5 Dropping a Collection

This example shows how to drop a collection.

```
export function openAndDropCollection(collectionName) {

    // look the collection up
    const col = soda.openCollection(collectionName);
```

```
    if (col === null) {
        throw new Error (`No such collection ${collectionName}`);
    }

    // drop the collection - POTENTIALLY DANGEROUS
    col.drop();
}
```

Creating Documents with SODA for In-Database JavaScript

Creation of documents by SODA for In-Database JavaScript is described.

The `SodaDocument` class represents SODA documents. Although its focus is on JSON documents, it supports other content types as well. A `SodaDocument` stores both the actual document's contents as well as metadata.

JavaScript is especially well-suited to work with JSON by design, giving it an edge over other programming languages.

Here is an example of a simple JSON document:

```
// Create a JSON document (based on the HR.EMPLOYEES table for employee 100)
const doc = {
    "_id": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
};
```

> **Note:**
>
> In SODA, JSON content must conform to RFC 4627.

`SodaDocument` objects can be created in three ways:

- As a result of `sodaDatabase.createDocument()`. This is a proto-`SodaDocument` object usable for SODA insert and replace methods. The `SodaDocument` will have content and media type components set.
- As a result of a read operation from the database, such as calling `sodaOperation.getOne()`, or from `sodaDocumentCursor.getNext()` after a `sodaOperation.getCursor()` call. These return complete `SodaDocument` objects containing the document content and attributes, such as media type.
- As a result of `sodaCollection.insertOneAndGet()`, `sodaOperation.replaceOneAndGet()`, or `sodaCollection.insertManyAndGet()` methods. These return `SodaDocuments` that contain all attributes except the document content itself. They are useful for finding document attributes such as system generated keys, and versions of new and updated documents.

A document has these components:

- Key
- Content
- Version
- Media type (`"application/json"` for JSON documents)

The document's content consists of all the fields representing the information the application needs to store plus an `_id` field. This field is either provided by the user or injected by Oracle if omitted. If omitted, Oracle adds a random value with a length of 12 bytes.

The document's key is a hex-encoded representation of the document's `_id` column. It is automatically calculated and cannot be changed. The key is often used when building operations such as finds, replaces, and removes, with `key()` and `keys(...)` methods. These operations are discussed in later sections.

Example 8-6 Creating SODA Documents

```
export function createJSONDoc() {

  // define the document's contents
  const payload = {
    "_id ": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
  };

  // Create a SODA document.
  // Notice that neither key nor version are populated. They will be as soon
  // as the document is inserted into a collection and retrieved.
  const doc = soda.createDocument(payload);
  console.log(`
    ----------------------------------------
    SODA Document using default key
    content (select fields):
    - _id          ${doc.getContent()._id}
    - job_id       ${doc.getContent().job_id}
    - first name   ${doc.getContent().first_name}
    media type:    ${doc.mediaType}
    version   :    ${doc.version}
    key            ${doc.key}`
  );
}
```

Creating `SodaDocument` instances as shown in this example is the exception rather than the norm. In most cases, developers use `SodaCollection.insertOne()` or `SodeCollection.insertOneAndGet()`. The use of `SodaCollection.insertOne()` is demonstrated in Example 8-7. Multiple documents can be created using `sodaCollection.insertMany()`.

Inserting Documents into Collections with SODA for In-Database JavaScript

`SodaCollection.insertOne()` or a related call such as `sodaCollection.insertOneAndGet()` offers convenient ways to add documents to a collection. These methods create document keys automatically, unless the collection is configured with client-assigned keys and the input document provides the key, which is not recommended for must users.

`SodaCollection.insertOne()` simply inserts the document into the collection, whereas `SodaCollection.insertOneAndGet()` additionally returns a result document. The resulting document contains the document key and any other generated document components, except for the actual document's content (this is done to improve performance).

Both methods automatically set the document's version, unless the collection has been created with custom metadata. Custom metadata might not include all the default metadata. When querying attributes not defined by the collection a null value is returned.

> **Note:**
>
> If you want the input document to *replace* the existing document instead of causing an exception, see Saving Documents into Collections with SODA for In-Database JavaScript.

Example 8-7 Inserting a SODA Document into a Collection

This example demonstrates how to insert a document into a collection using `SodaCollection.insertOne()`.

```
export function insertOneExample() {

  // define the document's contents
  const payload = {
    "_id": 100,
    "job_id": "AD_PRES",
    "last_name": "King",
    "first_name": "Steven",
    "email": "SKING",
    "manager_id": null,
    "department_id": 90
  };

  // create or open the collection to hold the document
  const col = soda.createCollection("MyCollection");

  col.insertOne(payload);
}
```

Example 8-8 Inserting an Array of Documents into a Collection

This example demonstrates the use of `SodaCollection.insertMany()` to insert multiple documents with one command. The example essentially translates the relational table `HR.employees` into a collection.

```
export function insertManyExample() {

  // select all records from the hr.employees table into an array
  // of JavaScript objects in preparation of a call to insertMany
  const result = session.execute(
    `SELECT
      employee_id "_id",
      first_name "firstName",
      last_name "lastName",
      email "email",
      phone_number "phoneNumber",
      hire_date "hireDate",
      job_id "jobId",
      salary "salary",
      commission_pct "commissionPct",
      manager_id "managerId",
      department_id "departmentId"
    FROM
      hr.employees`,
    [],
    { outFormat: oracledb.OUT_FORMAT_OBJECT }
  );

  // create the collection and insert all employee records
  collection = soda.createCollection('employeesCollection');
  collection.insertMany(result.rows);

  // the MLE JavaScript SQL driver does not auto-commit
  session.commit();
}
```

Saving Documents into Collections with SODA for In-Database JavaScript

You use `SodaCollection.save()` and `saveAndGet()` to save documents into collections.

These methods are similar to methods `insertOne()` and `insertOneAndGet()` except that, if the collection is configured with client-assigned document keys, and the input document provides a key that already identifies a document in the collection, then the input document *replaces* the existing document. In contrast, methods `insertOne()` and `insertOneAndGet()` throw an exception in that case.

SODA for In-Database JavaScript Read and Write Operations

The primary way you specify read and write operations (other than insert and save) is to use methods provided by the `SodaOperation` class. You can chain together `SodaOperation` methods to specify read or write operations against a collection.

Nonterminal `SodaOperation` methods return the same object on which they are invoked, allowing them to be chained together.

A *terminal* `SodaOperation` method always appears at the end of a method chain to execute the operation.

> **Note:**
>
> A `SodaOperation` object is an internal object. You should not directly modify its properties.

Unless the SODA documentation for a method says otherwise, you can chain together any nonterminal methods and you can end the chain with any terminal method. However, not all combinations make sense. For example, it does not make sense to chain method `version()` together with a method that does not uniquely identify the document, such as `keys()`.

Table 8-1 Overview of Nonterminal Methods for Read Operations

Method	Description
`key()`	Find a document that has the specified document key.
`keys()`	Find documents that have the specified document keys.
`filter()`	Find documents that match a filter specification (a query-by-example expressed in JSON).
`version()`	Find documents that have the specified version. This is typically used with `key()`.
`headerOnly()`	Exclude document content from the result.
`skip()`	Skip the specified number of documents in the result.
`limit()`	Limit the number of documents in the result to the specified number.

Table 8-2 Overview of Terminal Methods for Read Operations

Method	Description
`getOne()`	Create and execute an operation that returns at most one document. For example, an operation that includes an invocation of nonterminal method `key()`.
`getCursor()`	Get a cursor over read operation results.
`count()`	Count the number of documents found by the operation.
`getDocuments()`	Gets an array of documents matching the query criteria.

Table 8-3 Overview of Terminal Methods for Write Operations

Method	Description
`replaceOne()`	Replace one document.
`replaceOneAndGet()`	Replace one document and return the result document.
`remove()`	Remove documents from a collection.

> **See Also:**
>
> - Node-oracledb Documentation for more details about the `SodaOperations` class.
> - SODA Restrictions (Reference) for information about SODA restrictions.

Finding Documents in Collections with SODA for In-Database JavaScript

To find documents in a collection, you invoke `SodaCollection.find()`. It creates and returns a `SodaOperation` object which is used via method chaining with nonterminal and terminal methods.

To execute the query, obtain a cursor for its results by invoking `SodaOperation.getCursor()`. Then use the cursor to visit each document in the result list. This is illustrated by Example 8-1 and other examples. It is important not to forget to close the cursor, to save resources.

However, this is not the typical workflow when searching for documents in a collection. It is more common to chain multiple methods provided by the `SodaOperation` class together.

Example 8-9 Finding a Document by Key

This example shows how to look up a document by its key using the methods `find()`, `key()`, and `getOne()`.

```
export function findDocByKey(searchKey){

  const collectionName = 'MyCollection';

  // open the collection in preparation of a document lookup
  const col = soda.openCollection(collectionName);
  if (col === null){
    throw new Error(`${collectionName} does not exist`);
  }

  try{
    // perform a lookup of a document with the key provided as a
    // parameter to this function. Keys are like primary keys,
    // the lookup therefore can only return 1 document max
    const doc = col.find()
                   .key(searchKey)
                   .getOne();
    console.log(`
      document found for key ${searchKey}
      contents: ${doc.getContentAsString()}`
    );
  } catch(err){
      throw new Error(
        `error retrieving document with key ${searchKey} (${err})`
      );
  }
}
```

> **Note:**
>
> Keys need to be enclosed in quotation marks even if they should be in numeric format.

In case the search for a given key fails, the database throws an ORA-01403 (no data found) exception. It is good practice to handle exceptions properly. In this example, the caller of the function has the responsibility to ensure the error is trapped and dealt with according to the industry's best-known methods.

Example 8-10 Looking up Documents Using Multiple Keys

This example uses the methods `find()`, `keys()`, `getCursor()`, and `getNext()` to search for multiple keys provided in an array.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function findDocByKeys(searchKeys){

  if(!Array.isArray(searchKeys)){
    throw new Error('please provide an array of search keys');
  }

  // open a collection in preparation of a document lookup
  const col = soda.openCollection('employeesCollection');
  if (col === null){
    throw new Error('employeesCollection does not exist');
  }

  try{
    // perform a lookup of a set of documents using
    // the "keys" array provided
    const docCursor =
      col.find()
         .keys(searchKeys)
         .getCursor();

    let doc
      while((doc = docCursor.getNext())){
        console.log(`
          document found for key ${doc.key}
          contents: ${doc.getContentAsString()}`
        );
      }
      docCursor.close();
  } catch(err){
      // there is no error thrown if one/all of the keys aren't found
      // this error handler is generic
      throw new Error(
        `error retrieving documents with keys ${searchKeys} (${err})`
      );
  }
}
```

Rather than failing with an error, the `find()` operation simply doesn't return any data for a key not found in a collection. If none of the keys are found, nothing is returned.

Example 8-11 Using a QBE to Filter Documents in a Collection

This example uses `filter()` to locate documents in a collection. The nonterminal `SodaOperation.filter()` method provides a powerful way to filter JSON documents in a collection, allowing for complex document queries and ordering of JSON documents. Filter specifications can include comparisons, regular expressions, logical and spatial operators, among others.

The search expression defined in `filterCondition` matches all employees with an employee ID greater than `110` working in department `30`.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function findDocByFiltering(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error(`employeesCollection does not exist`);
  }

  // find all employees with an employee_id > 100 and
  // last name beginning with M
  const filterCondition = {
    "$and": [
      { "lastName": { "$upper": { "$startsWith": "M" } } },
      { "_id": { "$gt": 100 } }
    ]
  };

  try{

    // perform the lookup operation using the QBE defined earlier
    const docCursor = col.find()
                         .filter(filterCondition)
                         .getCursor();
    let doc;
    while ((doc = docCursor.getNext())){
      console.log(`
        ----------------------------------
        document found matching the search criteria
        - key:           ${doc.key}
        - _id:           ${doc.getContent()._id}
        - name:          ${doc.getContent().lastName}`
      );
    }

    docCursor.close();
  } catch(err){
      throw new Error(`error looking up documents using a QBE: ${err}`);
```

```
  }
}
```

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for an introduction to SODA filter specifications
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for reference information about SODA filter specifications

Example 8-12 Using skip() and limit() in a Pagination Query

If the number of rows becomes too large, you may choose to paginate and or limit the number of documents returned. This example demonstrates using `skip()` and `limit()` in this type of circumstance.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function paginationExample(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error ('employeesCollection does not exist, aborting');
  }

  // find all employees with an employee_id > 100 and
  // last name beginning with E
  const filterCondition = {
    "$and": [
      { "lastName": { "$upper": { "$startsWith": "M" } } },
      { "_id": { "$gt": 100 } }
    ]
  };

  try{

    // perform the lookup operation using the QBE, skipping the first
    // 5 documents and limiting the result set to 10 documents
    const docCursor =
      col.find()
         .filter(filterCondition)
         .skip(5)
         .limit(10)
         .getCursor();
    let doc;
    while ((doc = docCursor.getNext())){
      console.log(`
        ------------------------------------
```

```
        document found matching the search criteria
        - key:           ${doc.key}
        - employee id:   ${doc.getContent().employeeId}`
      );
    }

    docCursor.close();
  } catch(err){
      throw new Error(
        `error looking up documents by QBE (${err})`
      );
  }
}
```

Example 8-13 Specifying Document Versions

This example uses the nonterminal `version()` method to specify a particular document version. This is useful for implementing optimistic locking, when used with the terminal methods for write operations.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function versioningExample(searchKey, version){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection("employeesCollection");

  try{
    // perform a lookup of a document using the provided key and version
    const doc = col
      .find()
      .key(searchKey)
      .version(version)
      .getOne();
    console.log(`
      document found for key ${doc.key}
      contents: ${doc.getContentAsString()}`
    );
  } catch(err){
      throw new Error(
        `${err} during lookup. Key: ${searchKey}, version: ${version}`
      );
  }
}
```

If SODA cannot find the document matching the key and version tag, an `ORA-01403: no data found` error is thrown.

Example 8-14 Counting the Number of Documents Found

This example shows how to count the number of documents found in a collection using the `find()`, `filter()`, and `count()` methods. The `filter()` expression limits the result to all employees working in department `30`.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function countingExample(){

  // open a collection in preparation of a document
  // lookup. This particular collection contains all the
  // rows from the HR.employees table converted to SODA
  // documents.
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error('employeesCollection does not exist');
  }

  try{

    // perform a lookup operation identifying all employees working
    // in department 30, limiting the result to headers only
    const filterCondition = {"departmentId": 30};
    const numDocs = col.find()
                       .filter(filterCondition)
                       .count();
    console.log(`there are ${numDocs} documents matching the filter`);
  } catch(err){
      throw new Error(
        `No document found in 'employeesCollection' matching the filter`
      );
  }
}
```

Replacing Documents in a Collection with SODA for In-Database JavaScript

To replace the content of one document in a collection with the content of another, you start by looking up the document to be modified using its key. Because `SodaOperation.key()` is a nonterminal operation, the easiest way to replace the contents is to chain `SodaOperation.key()` to `SodaOperation.replaceOne()` or `SodaOperation.replaceOneAndGet()`.

`SodaOperation.replaceOne()` merely replaces the document, whereas `SodaOperation.replaceOneAndGet()` replaces it and provides the resulting new document to the caller.

The difference between `SodaOperation.replace()` and `SodaOperation.save()` is that the latter performs an insert in case the key doesn't already exist in the collection. The replace operation requires an existing document to be found by the lookup via the `SodaOperation.key()` method.

> **Note:**
>
> Some version-generation methods generate hash values of the document content. In such a case, if the document content does not change then neither does the version.

Example 8-15 Replacing a Document in a Collection and Returning the Result Document

This example shows how to replace a document in a collection, returning a reference to the changed document. Let's assume that employee 206 has been given a raise of 100 monetary units. Using the SODA API you can update the salary as follows:

```
export function replaceExample(){

  // open employeesCollection in preparation of the update
  const col = soda.openCollection('employeesCollection');
  if (col === null){
    throw new Error("'employeesCollection does not exist");
  }

  try{
    // look up employeeId 206 using a QBE and get the document.
    // Since the documents are inserted into the collection based
    // on the HR.employees table, it is certain that there is at
    // most 1 document with employeeId 206
    const employeeDoc = col
                        .find()
                        .filter({"_id": 206})
                        .getOne();

    // get the document's actual contents/payload
    employee = employeeDoc.getContent();

    // currently it is not possible to include the _id together with
    // the replacement payload. This means existing _id must be deleted.
    // The document, once replaced in the collection, will have its
    // _id injected from the target document
    delete employee_id;

    // increase the salary
    employee.salary += 100;

    // save the document back to the collection. Note that you need
    // to provide the document's key rather than a QBE or else an
    // ORA-40734: key for the document to replace must be specified
    // using the key attribute error will be thrown
    const resultDoc = col
                      .find()
                      .key(employeeDoc.key)
                      .replaceOneAndGet(employee);

    // print some metadata (note that content is not returned for
    // performance reasons)
    console.log(`Document updated successfully:
    - key:           ${resultDoc.key}
    - version:       ${resultDoc.version}`);

  } catch(err){
      console.log(`error modifying employee 206's salary: ${err}`);
  }
}
```

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

> **Note:**
>
> Trying to read the changed contents will result in an error as the actual document's contents aren't returned, for performance reasons.

Removing Documents from a Collection with SODA for In-Database JavaScript

Removing documents from a collection is similar to replacing. The first step is to perform a lookup operation, usually based on the document's key or by using a search expression in `SodaOperation.filter()`. The call to `SodaOperation.remove()` is a terminal operation, in other words the last operation in the chain.

Example 8-16 Removing a Document from a Collection Using a Document Key

This example removes the document whose document key is `"100"`.

```
export function removeByKey(searchKey){

  // open MyCollection
  const col = soda.openCollection("MyCollection");
  if(col === null){
    throw new Error("'MyCollection' does not exist");
  }

  // perform a lookup of the document about to be removed
  // and ultimately remove it
  const result = col
                 .find()
                 .key(searchKey)
                 .remove();
  if(result.count === 0){
    throw new Error(
      `failed to delete a document with key ${searchKey}`
    );
  }
}
```

Example 8-17 Removing JSON Documents from a Collection Using a Filter

This example uses a filter to remove the JSON documents whose `department_id` is `70`. It then prints the number of documents removed.

```
export function removeByFilter(){

  // open the collection
  const col = soda.openCollection("MyCollection");
  if(col === null){
    throw new Error("'MyCollection' does not exist");
  }
```

```
  // perform a lookup based on a filter expression and remove
  // the documents matching the filter
  const result = col
                 .find()
                 .filter({"_id": 100})
                 .remove();

  console.log(`${result.count} documents deleted`);
}
```

Indexing the Documents in a Collection with SODA for In-Database JavaScript

Indexes can speed up data access, regardless of whether you use the NoSQL style SODA API or a relational approach. You index documents in a SODA collection using `SodaCollection.createIndex()`. Its `IndexSpec` parameter is a textual JSON index specification.

Existing indexes can be dropped using `SodaCollection.dropIndex()`.

A JSON search index is used for full-text and ad hoc structural queries, and for persistent recording and automatic updating of JSON data-guide information.

> **See Also:**
>
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for an overview of using SODA indexing
> - *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about SODA index specifications
> - *Oracle Database JSON Developer's Guide* for information about JSON search indexes
> - *Oracle Database JSON Developer's Guide* for information about persistent data-guide information as part of a JSON search index

Example 8-18 Creating a B-Tree Index for a JSON Field with SODA for In-Database JavaScript

This example creates a B-tree non-unique index for numeric field `department_id` of the JSON documents in collection `employeesCollection` (created in Example 8-8).

```
export function createBTreeIndex(){

  // open the collection
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // define the index...
  const indexSpec = {
```

```
    "name": "DEPARTMENTS_IDX",
    "fields": [
      {
        "path": "departmentId",
        "datatype": "number",
        "order": "asc"
      }
    ]
  };

  //... and create it
  try{
    col.createIndex(indexSpec);
  } catch(err){
      throw new Error(
        `could not create the index: ${err}`
      )
  }
}
```

Example 8-19 Creating a JSON Search Index with SODA for In-Database JavaScript

This example shows how to create a JSON search index for indexing the documents in collection `employeesCollection` (created in Example 8-8). It can be used for ad hoc queries and full-text search (queries using QBE operator `$contains`). It automatically accumulates and updates data-guide information about your JSON documents (aggregate structural and type information). The index specification has only field `name` (no *field* fields unlike the B-tree index in Example 8-18).

```
export function createSearchIndex(){

  // open the collection
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // define the index properties...
  cost indexSpec = {
    "name": "SEARCH_AND_DATA_GUIDE_IDX",
    "dataguide": "on",
    "search_on": "text_value"
  }

  //...and create it
  try{
    col.createIndex(indexSpec);
  } catch(err){
      throw new Error(
        `could not create the search and Data Guide index: ${err}`
      );
  }
}
```

If you only wanted to speed up ad hoc (search) indexing, you should specify a value of "off" for field `dataguide`. The `dataguide` indexing feature can be turned off in the same way if it is not required.

Example 8-20 Dropping an Index with SODA for In-Database JavaScript

This example shows how you can drop an existing index on a collection using `SodaCollection.dropIndex()` and the `force` option.

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

```
export function dropIndex(indexName){

  // open the collection
  const col = soda.openCollection("employeesCollection");
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // drop the index
  const result = col.dropIndex(indexName, {"force": true});
  if(!result.dropped){
    throw `Could not drop SODA index '${indexName}'`;
  }
}
```

`SodaCollection.dropIndex()` returns a result object containing a single field: `dropped`. Its value is `true` if the index has been dropped, otherwise its value is `false`. The method succeeds either way.

An optional parameter object can be supplied to the method. Setting `force` to `true` forces dropping of a JSON index if the underlying Oracle Database domain index does not permit normal dropping.

Getting a Data Guide for a Collection with SODA for In-Database JavaScript

A data guide is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents. They provide great insights into JSON documents and are invaluable for getting an overview of a data set.

You can create a data guide using `SodaCollection.getDataGuide()`. To get a data guide in SODA, the collection must be JSON-only and have a JSON search index where the `"dataguide"` option is `"on"`. Data guides are returned from `sodaCollection.getDataGuide()` as JSON content in a `SodaDocument`. The data guide is inferred from the collection as it currently is. As a collection grows and documents change, a new data guide is returned each subsequent time `getDataGuide()` is called.

Example 8-21 Generating a Data Guide for a Collection

This example gets a data guide for the collection `employeesCollection` (created in Example 8-8) using the method `getDataGuide()` and then prints the contents as a string using the method `getContentAsString()`.

```
export function createDataGuide(){

  // open the collection
```

```
  const col = soda.openCollection('employeesCollection');
  if(col === null){
    throw new Error("'employeesCollection' does not exist");
  }

  // generate a Data Guide (requires the Data Guide index)
  const doc = col.getDataGuide();
  console.log(doc.getContentAsString());
}
```

The data guide can provide interesting insights into a collection, including all the fields and their data types. Although the Data Guide for `employeesCollection` may already be familiar to readers of this chapter, unknown JSON documents can be analyzed conveniently this way. The previous code block prints the following Data Guide to the screen:

```
{
  "type": "object",
  "o:length": 1,
  "properties": {
    "_id": {
      "type": "id",
      "o:length": 24,
      "o:preferred_column_name": "DATA$_id"
    },
    "email": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$email"
    },
    "jobId": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$jobId"
    },
    "salary": {
      "type": "number",
      "o:length": 8,
      "o:preferred_column_name": "DATA$salary"
    },
    "hireDate": {
      "type": "string",
      "o:length": 32,
      "o:preferred_column_name": "DATA$hireDate"
    },
    "lastName": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$lastName"
    },
    "firstName": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$firstName"
    },
    "managerId": {
```

```
      "type": "string",
      "o:length": 4,
      "o:preferred_column_name": "DATA$managerId"
    },
    "employeeId": {
      "type": "number",
      "o:length": 4,
      "o:preferred_column_name": "DATA$employeeId"
    },
    "phoneNumber": {
      "type": "string",
      "o:length": 16,
      "o:preferred_column_name": "DATA$phoneNumber"
    },
    "departmentId": {
      "type": "string",
      "o:length": 4,
      "o:preferred_column_name": "DATA$departmentId"
    },
    "commissionPct": {
      "type": "string",
      "o:length": 32,
      "o:preferred_column_name": "DATA$commissionPct"
    }
  }
}
```

Handling Transactions with SODA for In-Database JavaScript

Unlike the client-side JavaScript SQL driver, the MLE JavaScript SQL driver does not provide an `autoCommit` feature. You need to commit or roll your transactions back, either in the PL/SQL layer in case of module calls, or directly in the JavaScript code by calling `connection.commit()` or `connection.rollback()`.

> **Caution:**
>
> If any uncommitted operation raises an error, and you do not explicitly roll back the transaction, the incomplete transaction might leave the relevant data in an inconsistent state (uncommitted, partial results).

Creating Call Specifications Involving the SODA API

Earlier in this chapter, in the section *Getting Started with SODA for In-Database JavaScript*, an example showing how to invoke the MLE SODA API using an inline call specification is included. The following short example demonstrates how to use SODA in MLE modules.

Example 8-22 Use SODA for In-Database JavaScript

See Example 8-8 for details about how to create `employeesCollection`, used in this example.

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

```
/**
 * Example for a private function used to open and return a SodaCollection
 *
 * @param {string} collectionName the name of the collection to open
 * @returns {SodaCollection} the collection handle
 * @throws Error if the collection cannot be opened
 */
function openAndCheckCollection(collectionName){

  const col = soda.openCollection(collectionName);
  if(col === null){
    throw new Error(`invalid collection name: ${collectionName}`);
  }

  return col;
}

/**
 * Top-level (public) function demonstrating how to use a QBE to
 * filter documents in a collection.
 *
 * @param {number} departmentId the numeric department ID
 * @returns {number} the number of employees found in departmentId
 */
export function simpleSodaDemo(departmentId){

  if(departmentId === undefined || isNaN(departmentId)){
    throw new Error('please provide a valid numeric department ID');
  }

  const col = openAndCheckCollection('employeesCollection');

  const numDocs = col.find()
                     .filter({"departmentId": departmentId})
                     .count();

  return numDocs;
}
/
```

After the module has been created you need to create the call specification. The module features a single public function, so a standalone function should suffice:

```
CREATE OR REPLACE FUNCTION simple_soda_demo(
  "departmentId" NUMBER
) RETURN NUMBER
AUTHID current_user
AS MLE MODULE end_to_end_demo
SIGNATURE 'simpleSodaDemo';
/
```

Now everything is in place to call the function:

```
select simple_soda_demo(30);
```

Result:

```
SIMPLE_SODA_DEMO(30)
--------------------
                   6
```

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!
```

10

MLE Security

MLE utilizes a number of methods to support good security practices. This includes enforcing runtime state isolation, system and object privileges, and providing monitoring options.

Topics

System and Object Privileges Required for Working with JavaScript in MLE

Depending on the project's requirements, different privileges can be granted to users and or roles, allowing them to interact with JavaScript in the database.

Administrators should review application requirements carefully and only grant the minimum number of privileges necessary to users. This is especially true for system privileges, which are very powerful and should only be granted to trusted users.

The minimum privilege required to work with MLE JavaScript code is the right to execute JavaScript code in the database. MLE distinguishes between dynamic MLE execution based on `DBMS_MLE` and MLE execution using MLE modules and environments.

Creating stored code in JavaScript requires additional privileges to create JavaScript schema objects in your own schema.

The most powerful privileges available in MLE allow super-users to create, alter, and drop MLE schema objects in any schema, not just their own. As with all privileges in Oracle Database, those with `ANY` in their name are most powerful and should only be granted to trusted users if deemed absolutely necessary.

> **Note:**
>
> Object privileges on modules and environments do not grant access to an application, for example, the combination of source code and user context defined by a call specification (or through `DBMS_MLE`). This is achieved by granting access to the procedure or function object of the call specification.

> **See Also:**
>
> - Necessary Privileges for Creating MLE Modules and Environments in ANY Schema for more about handling system privileges
> - *Oracle Database Security Guide* for more information about privileges in the Oracle Database

Topics

Necessary Privileges for the Execution of JavaScript Code

Before you can execute any JavaScript code in your own schema, the following object grant must have been issued to your user account:

```
GRANT EXECUTE ON JAVASCRIPT TO <role | user>
```

The `EXECUTE ON JAVASCRIPT` privilege does not include dynamic execution of JavaScript using `DBMS_MLE`. If you wish to make use of `DBMS_MLE`, an additional privilege is required:

```
GRANT EXECUTE DYNAMIC MLE TO <role | user>
```

Necessary Privileges for Using the NoSQL API

In cases where MLE JavaScript code references the Simple Oracle Document Access (SODA), the `SODA_APP` role must be granted to the user or role:

```
GRANT SODA_APP <role | user>
```

Necessary Privileges for Creating MLE Schema Objects

If you wish to create MLE modules and environments in your own schema, further system privileges are required:

```
GRANT CREATE MLE TO <role | user>
```

In case any MLE module is to be exposed to the database's SQL and PL/SQL layers in the form of call specifications, you also require the right to create PL/SQL procedures:

```
GRANT CREATE PROCEDURE TO <role | user>
```

It is highly likely that you will require further system privileges, depending on your use case, to create additional schema objects such as tables, indexes, and sequences. Beginning with Oracle Database 23ai, the `DB_DEVELOPER_ROLE` role allows administrators to grant the necessary privileges to developers in their local development databases quickly. The role can be granted as shown in the following snippet:

```
GRANT DB_DEVELOPER_ROLE TO <role | user>
```

> **See Also:**
>
> *Oracle Database Security Guide* for more information about the `DB_DEVELOPER_ROLE` role

Necessary Privileges for Creating MLE Modules and Environments in ANY Schema

Additional privileges can be granted to power users and administrators, allowing them to create, alter, and drop MLE schema objects in *any* schema.

```
GRANT CREATE ANY MLE TO <role | user>
GRANT DROP ANY MLE TO <role | user>
GRANT ALTER ANY MLE TO <role | user>
```

As with all privileges in Oracle Databases featuring `ANY` in their name, these are very powerful and should only be granted after a thorough investigation to trusted users. For this reason, only the `DBA` role and the `SYS` account have been granted these privileges. The use of these system privileges is audited by the `ORA_SECURECONFIG` audit policy.

To create MLE call specifications in schemas other than your own requires the right to `CREATE ANY PROCEDURE` to be granted as well:

```
GRANT CREATE ANY PROCEDURE TO <role | user>
```

Just like the previously listed system privileges, `CREATE ANY PROCEDURE` is audited by the same audit policy, `ORA_SECURECONFIG`.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about the `ORA_SECURECONFIG` audit policy

Necessary Privileges for Post-Execution Debugging

It is possible to allow other database users to collect debug information for MLE modules they don't own. By default, MLE owners can use post-execution debugging on their own MLE modules without specific grants. It is possible to grant the ability to collect debug information to a different role or user, allowing them to use post-execution debugging of JavaScript code on your behalf as the module owner:

```
GRANT COLLECT DEBUG INFO ON <module> TO <role | user>
```

> **Note:**
>
> You can elect to grant the execute privilege on MLE module calls created as PL/SQL code with definer's rights to users in other schemas. In this case, there is no need to grant other users any additional privileges.

> **Note:**
>
> Object privileges on modules and environments do not grant access to an application, for example, the combination of source code and user context defined by a call specification (or through `DBMS_MLE`). This is achieved by granting access to the procedure or function object of the call specification.

> **See Also:**
>
> Post-Execution Debugging of MLE JavaScript Modules for more information on post-execution debugging

Security Considerations for MLE

Besides the use of account privileges, MLE employs several other methods to ensure a high level of security.

Topics

- MLE_PROG_LANGUAGES Initialization Parameter

MLE_PROG_LANGUAGES Initialization Parameter

A new initialization parameter, `MLE_PROG_LANGUAGES`, allows administrators to enable and disable Multilingual Engine completely or selectively enable certain languages. It takes the values `ALL`, `JAVASCRIPT`, or `OFF` and it can be set at multiple levels:

- Container Database (CDB)
- Pluggable Database (PDB)
- Database session

If the parameter is set to `OFF` at CDB level, it cannot be enabled at PDB or session level. The same logic applies for PDB and session level: if MLE is disabled at the PDB level, it cannot be enabled at session level.

> **Note:**
>
> In Oracle Database 23ai, MLE supports JavaScript as its sole language. Setting the parameter to `ALL` or `JAVASCRIPT` has the same effect.

> **Note:**
>
> Setting `MLE_PROG_LANGUAGES` to `OFF` prevents the execution of JavaScript code in the database, it does not prevent the creation or modification of existing code.

> **See Also:**
>
> *Oracle Database Reference* for more information about `MLE_PROG_LANGUAGES`

Execution Contexts

When executing JavaScript code in the database, MLE uses execution contexts to isolate runtime state such as global variables and other important information. Execution contexts are created implicitly when using modules and environments and explicitly when using `DBMS_MLE`.

Regardless of the choice of JavaScript invocation, execution contexts are designed to prevent information leak.

The scope of JavaScript state never exceeds the lifetime of a database session. As soon as the session ends, either gracefully or forcefully, session state is discarded. If state needs to be

preserved between sessions, you must persist it by storing it in a schema. If needed, state can be discarded by calling `DBMS_SESSION.reset_package()`.

As an additional security measure, you can optionally specify the use of a restricted execution context, which disallows access to the database state. The `PURE` keyword is used in the creation of environments and in inline call specifications to indicate the use of a restricted context. An environment created using `PURE` can be referenced in module call specifications and using `DBMS_MLE`. 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.

See Also:

- About Restricted Execution Contexts for more information about the `PURE` keyword and restricted contexts
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_SESSION`

Runtime State Isolation

An MLE call specification is a PL/SQL unit referencing a function in an MLE module with an optional MLE environment attached. When you invoke a call specification in a session, the corresponding MLE module is loaded, the optional environment is applied, and the function specified in the call specification's signature clause is executed.

Before execution can begin, a corresponding execution context must be created (implicitly). Whether a new execution context is created or an existing context is reused depends on multiple factors, specifically:

- The MLE module referenced in the call specification
- The corresponding MLE environment
- The database user executing the call specification

Separate execution contexts are created to prevent information leak as well as undesired side effects such as global variables in a module being overwritten by accident.

With each invocation of a call specification, additional execution contexts are created. This is done so that modules cannot interfere with one another.

The main criteria for creating execution contexts in a user session are the MLE module name and the corresponding MLE environment. Call specifications referring to different combinations of MLE module and environment lead to different individual execution contexts being created.

Further separation between execution contexts is performed based on the user invoking the call specification.

Example 10-1 Runtime State Isolation Scenario

This example provides a sample scenario for runtime state isolation. Database user `USER1` creates the following MLE schema objects:

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

```
let id;      // global variable

export function doALotOfWork() {
  // a dummy function simulating a lot of work
  // the focus is on modifying a global variable

  id = 10;
}

export function getId() {

  return (id === undefined ? -1 : id)
}
/

CREATE OR REPLACE MLE ENV isolationEnv;

CREATE OR REPLACE PACKAGE context_isolation_package AS

  -- initialise runtime state
  procedure doALotOfWork as
    mle module isolationMod
    signature 'doALotOfWork()';

  -- access a global variable (part of session state)
  function getId return number as
    mle module isolationMod
    signature 'getId()';

  -- same function signature as before but referencing an environment
  function getIdwEnv return number as
    mle module isolationMod
    env isolationEnv
    signature 'getId()';
END;
/
```

When `USER1`, the owner of the MLE module, environment, and call specification (package), calls `context_isolation_package.doALotOfWork()`, the global variable (`id`) is initialized to `10`.

```
BEGIN
    context_isolation_package.doALotOfWork();
END;
/
```

Because `context_isolation_package.getId()` references the same MLE module and the same (default) environment as `context_isolation_package.doALotOfWork()`, the user's session has access to the global variable:

```
SELECT CONTEXT_ISOLATION_PACKAGE.getId;

     GETID
----------
        10
```

When the combination of user, MLE module, and environment change, a new execution context is created. Although `context_isolation_package.getIdwEnv()` references the same MLE module as `getID()` and the user doesn't change, the function cannot retrieve the value of the global variable from the previously created execution context:

```
SELECT CONTEXT_ISOLATION_PACKAGE.getIdwEnv;

 GETIDWENV
----------
        -1
```

A value of `-1` indicates that the global variable in the JavaScript module was found to be uninitialized.

If `USER1`, as the owner of the MLE call specification, grants the execute privilege on the package to another user, let's say `USER2`, a different execution context is created for `USER2` even though the same function is called:

```
GRANT EXECUTE ON CONTEXT_ISOLATION_PACKAGE TO user2;
```

When `USER2` tries to read the value of the `ID`, a new context is created and the return value indicating an uninitialized context is returned:

```
SELECT user1.CONTEXT_ISOLATION_PACKAGE.getid;

     GETID
----------
        -1
```

In this example, module and environment are identical between `USER1` and `USER2` as per the call specification. However, the fact that the function is called by a different user causes a new execution context to be created.

Database Security Model

The fewer privileges granted to program units, accounts, and roles, the less likely it is for them to be misused. As with every application, the principle of granting only the minimum number of necessary privileges should be followed. This is especially true in higher-tier environments like production. Technologies such as Privilege Analysis can be used to track down unnecessary privileges, allowing you to revoke them after careful regression testing.

Each MLE call specification is created within its own security context. The context includes information such as:

- The value of the `AUTHID` clause (definer or invoker)
- Whether or not privileges are inherited in invoker's rights calls
- Code Based Access Control
- Current user
- The qualified schema name
- Enabled Roles and Privileges in the absence of code based access control (CBAC) and invoker's rights

The combination of these attributes forms the security context of a code unit such as a MLE call specification or module. Note that no such security context exists for the JavaScript code stored in an MLE module.

PL/SQL allows you to easily change these attributes for each PL/SQL unit. A procedure can be executed with the invoker's rights or the definer's rights, roles can be attached to PL/SQL units, and cross-schema (execute) grants are commonplace. With each execution of a PL/SQL unit the security context may potentially change. This applies equally to MLE call specifications.

The situation is different with JavaScript code: the security context does not change for JavaScript-to-JavaScript calls. JavaScript functions do not have any notion of associated invoker's or definer's rights, or roles granted on the function itself. All of these apply only to (PL/SQL) call specifications.

JavaScript executed using `DBMS_MLE` is a little more strict when it comes to its security context. The combination of currently active user, roles/privileges, and schema in effect are recorded at the time the execution context is created by calling `DBMS_MLE.create_context()`. This combination must not change until the JavaScript code is executed and the context is removed, or else an error is thrown.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about Privilege Analysis

Considerations for Using MLE Call Specifications and Modules from Different Schemas

The same consideration that is used for other database applications written in, for example, PL/SQL apply for MLE JavaScript code as well. If a user is granted access to execute code from a schema other than their own, care needs to be taken to ensure the extent to which the code can use privileges of the calling user is appropriate.

Unlike PL/SQL, MLE JavaScript code stored in an MLE module is not associated with a particular set of roles, or any other notion of determining the security context in which the JavaScript code executes. From a high-level view, there are two important cases for cross-schema use of privileges:

1. `USER1` invokes a call specification located in `USER2`'s schema. The `AUTHID` clause of the call specification in `USER2`'s schema determines whether the code owned by `USER2`'s schema executes with the privileges of the invoker (`USER1`) or definer (`USER2`). In case of an invoker's rights call specification, potentially attached roles (CBAC) and the setting of `INHERIT PRIVILEGES` determine the active roles and privileges in addition to those granted by `USER1` by roles or direct grants.
2. `USER1` creates a call specification `CallSpec_A` for a module `Module_A` owned by `USER1`. `CallSpec_A` imports a JavaScript module `Module_B` owned by a different schema, `USER2`. The JavaScript code in `Module_B` is imported into an execution context created for `USER1`'s call specification `CallSpec_A`. The JavaScript code in `Module_B` executes with the same privileges as any other JavaScript code in this execution such as in `Module_A`. `USER1` must ensure that the code in `Module_B` is trustworthy and appropriate to execute with these privileges.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about roles in definer's rights and invoker's rights PL/SQL units

Auditing MLE Operations in Oracle Database

Auditing is the monitoring and recording of configured database actions. As with any other auditable operations in Oracle Database, the use of MLE-related system privileges can be recorded.

Oracle provides the `ORA_SECURECONFIG` audit policy with the database. Starting with Oracle Database 23ai, the audit policy includes the use of the following MLE system privileges:

- `CREATE ANY MLE`
- `ALTER ANY MLE`
- `DROP ANY MLE`

Administrators and security teams need to create and enable additional security policies if auditing the creation of MLE schema objects, including MLE modules, environments, and call specifications, is desired.

> **See Also:**
>
> *Oracle Database Security Guide* for more information about auditing in Oracle Database

JavaScript Security Best Practices

Details concerning the best practices when using features of MLE with JavaScript are described.

Topics

Using Bind Variables for Security and Performance

The MLE JavaScript SQL driver allows you to use string concatenation to build SQL commands, including the predicates used in queries and DML statements. It is strongly recommended to avoid this bad practice as it is a major source for SQL injection attacks. Not

only is the use of bind variables in SQL statements more secure than string concatenation but it is also more efficient as it allows the database to reuse the cursor in the shared pool.

If it is not possible to avoid the creation of dynamic SQL, ensure that you validate input to your code and scan for malicious content. The built-in `DBMS_ASSERT` package provides a wealth of functions designed to mitigate against SQL injection attacks. It does not offer complete protection but its use is very much recommended as it allows you to verify the following:

- The input string is a qualified SQL name
- The input string is an existing schema name
- The input string is a simple SQL name
- The input parameter string is a qualified SQL identifier of an existing SQL object

The use of bind variables for better security and scalability is not limited to a single programming language such as JavaScript, it equally applies to every development project using Oracle Database.

> **See Also:**
>
> - Server-Side JavaScript API Documentation for information about using bind variables with `mle-js-oracledb`
> - *Oracle Database Development Guide* for more details regarding bind variables and their impact on performance and security

Example 10-2 Using Bind Variables Rather than String Concatenation

In this example, the `SELECT` statement accepts a bind variable rather than concatenation the input variable, `managerID`, to the SQL command.

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

import oracledb from "mle-js-oracledb";

export function numEmployeesByManagerID(managerID) {

  const conn = oracledb.defaultConnection(managerID);
  const result = conn.execute(
    `SELECT count(*) FROM employees WHERE manager_id = :1`,
    [ managerID ]
  );

  return result.rows[0][0];
}
/
```

Example 10-3 Use DBMS_ASSERT to Verify Valid Input

In this example, the function `createTempTable()` creates a private temporary table to hold intermediate results from a batch process. The function takes a single argument: the name of

the temporary table to be created (minus the prefix). The function checks if the parameter passed to it is a valid SQL name.

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

import oracledb from "mle-js-oracledb";

export function createTempTable(tableName) {
  const conn = oracledb.defaultConnection();
  let result;
  let validTableName;

  try {
    result = conn.execute(
      `SELECT dbms_assert.qualified_sql_name(:tableName)`,
      [tableName]
    );
    validTableName = result.rows[0][0];
  } catch (err) {
    throw (`'${tableName}' is not a valid table name`);
    return;
  }

  result = conn.execute(
    `CREATE PRIVATE TEMPORARY TABLE ora\$ptt_${validTableName} (id number)`
  );
}
/
```

If the table name passed to the function passes the test, it is then used to create a private temporary table using the default `private_temp_table_prefix`.

Generic Database and PL/SQL Specific Security Considerations

Because all JavaScript code is accessed eventually via a PL/SQL call specification, it is important to understand the implications of using PL/SQL as well. The following concepts are of particular importance:

- The difference between invoker's rights and definer's rights
- Code Based Access Control (CBAC)
- The impact of `INHERIT PRIVILEGES` in invoker's rights code
- Role grants and direct grants, both object as well as system privileges

You should always aim to only require the minimum security privileges (object and system) for JavaScript code to execute. This is especially important when you consider the use of external third-party JavaScript code.

Administrators should consider the use of encryption for both data at rest as well as data in motion.

> **See Also:**
>
> - *Oracle Database Security Guide* for more information about generic database-related security aspects
> - *Oracle Database Transparent Data Encryption Guide* for information about encrypting data at rest using Transparent Data Encryption (TDE)

Supply Chain Security

Access to the rich community ecosystem is one of the advantages of using JavaScript in Oracle Database. Rather than creating functionality in-house and potentially duplicating effort, existing JavaScript can be used instead. While this is a convenient method for developing applications, it comes with certain risks.

In past years, the term supply chain attach has been used to describe the fact that certain popular open-source JavaScript modules have been abandoned by the original maintainers. Bad actors have taken some of these projects, becoming maintainers but only to inject malicious code into the source. The next time a project references such a compromised module, they incorporate the malicious code.

The same principles applied to client-side development apply to server-side development with MLE. Developers and security teams must be aware that code in the application executes with potentially elevated privileges. These can be abused by malicious code to compromise confidentiality, integrity, and availability properties of the application. For that reason, extra care must be taken to ensure third-party code is trustworthy and that the minimum number of privileges is granted to it. Many companies have a dedicated security team for vetting open-source modules prior to granting their approval to use them. At the very least, you should audit the JavaScript code that you are about to include in your project and document the result.

It is possible to lock a given version of an open-source module using a mechanism like the `package-lock.json` file so as not to get caught out if a new version of a module is distributed. Automatically pulling the latest version of an external code dependency is bad practice and should always be avoided.

In the case of JavaScript in MLE, JavaScript code executes with the database privileges that are in effect for the associated execution context. JavaScript code can retrieve and modify data stored in the database according to these privileges. Malicious code can leverage these privileges to modify the database in an inappropriate manner.

As a consequence, be sure to grant the privileges to create MLE modules carefully and only grant these in environments where they are essential. If possible, avoid granting the [`CREATE` | `ALTER` | `MODIFY`] `ANY` system privileges at all.

You should also review the `INHERIT PRIVILEGES` settings in the context of invoker's rights procedures. Once the settings for `INHERIT PRIVILEGES` are reviewed and secured according to industry best practice, consider the use of invoker's rights for MLE call specifications.

Additional higher levels of security for invoker's rights procedures can be achieved by implementing code based access control (CBAC). Using CBAC, developers can associate roles to PL/SQL units without having to elevate the privileges of the schema or invoker.

> **See Also:**
>
> *Oracle Database Security Guide* for details about the `INHERIT PRIVILEGES` privilege

Software Bill of Material

Every project relying on external code in projects is strongly encouraged to maintain a record of all software components (including versions) that are bundled in a deployed application artifact.

The software bill of material (SBOM) is the key tool to use when reacting swiftly to a newly published vulnerability is of utmost importance. Exploits are almost guaranteed to be used immediately after a vulnerability has been published. Knowing exactly which version of a third-party library is in use allows you to save crucial time in preparing a response.

In addition to storing the actual code, MLE modules feature a metadata field that can be used to store arbitrary metadata with the module. In particular, it can be used to store an SBOM that describes all JavaScript libraries bundled in the module. The field is not interpreted by the MLE runtime. Content and format are entirely up to you.

> **See Also:**
>
> MLE JavaScript Modules and Environments for more information about creating MLE modules and providing metadata to them

Using the Database to Store State

Applications written using MLE JavaScript code should not deviate from established patterns such as storing application state in tables. This allows you to make the best use of the rich number of security features available for Oracle Database.

In particular, you should not rely on JavaScript state that exceeds the boundaries of one stored procedure or function call.

Oracle Database has great support for JSON, offering both a relational as well as a NoSQL API. The database's JSON API is a natural candidate for MLE JavaScript code to store state. Storing state in Oracle Database provides a better programming model than application state, especially when it come to data persistence and transactional consistency.

> **See Also:**
>
> *Oracle Database JSON Developer's Guide* for information about using JSON with Oracle Database

Example 10-4 Using Bind Variables Rather than String Concatenation

In this example, the `SELECT` statement accepts a bind variable rather than concatenation the input variable, `managerID`, to the SQL command.

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

import oracledb from "mle-js-oracledb";

export function numEmployeesByManagerID(managerID) {

  const conn = oracledb.defaultConnection(managerID);
  const result = conn.execute(
    `SELECT COUNT(*) FROM employees WHERE manager_id = :1`,
    [ managerID ]
  );

  return result.rows[0][0];
}
/
```

Example 10-5 Use DBMS_ASSERT to Verify Valid Input

In this example, the function `createTempTable()` creates a private temporary table to hold intermediate results from a batch process. The function takes a single argument: the name of the temporary table to be created (minus the prefix). The function checks if the parameter passed to it is a valid SQL name.

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

import oracledb from "mle-js-oracledb";

export function createTempTable(tableName) {
  const conn = oracledb.defaultConnection();
  let result;
  let validTableName;

  try {
    result = conn.execute(
      `SELECT dbms_assert.qualified_sql_name(:tableName)`,
      [tableName]
    );
    validTableName = result.rows[0][0];
  } catch (err) {
    throw (`'${tableName}' is not a valid table name`);
    return;
  }

  result = conn.execute(
    `CREATE PRIVATE TEMPORARY TABLE ora\$ptt_${validTableName} (id number)`
  );
}
/
```

If the table name passed to the function passes the test, it is then used to create a private temporary table using the default `private_temp_table_prefix`.

Disabling Multilingual Runtime

In the case where a security vulnerability is detected in JavaScript code, you can prevent JavaScript code from execution by disabling the JavaScript runtime. Setting the initialization parameter `MLE_PROG_LANGUAGES` to `OFF` does not stop the database from accepting new code (such behavior prevents the implementation of a code fix) but it does stop anyone from executing JavaScript code.

Applications should be written with that option in mind. Once the MLE runtime is disabled, an error is thrown. Rather than showing the raw error to the end user, a more accessible error message should be created.

Although JavaScript does not have a specific lockdown feature, using the `MLE_PROG_LANGUAGES` parameter allows you to disable the MLE runtime at the session, PDB (lockdown profiles operate at this level), or CDB level. The `COMMON_SCHEMA_ACCESS` feature bundle in the lockdown profile can be used to disable MLE DDL.

MLE Security Examples

Example scenarios are used to demonstrate security features used by MLE. The examples use a varying degree of separation between MLE modules, environments, and the necessary grants to enable the utilized functionality.

Note that the examples are not fully usable on their own. The actual JavaScript code is not as important as the application's structure, such as:

- The schemas in which the code is located
- The call specification's syntax
- The roles and privileges granted

Topics

Business Logic Stored in MLE Modules

In this scenario, a user provides functionality implemented in JavaScript that is bound to a particular schema and relies on being executed as a particular user with certain privileges.

This scenario covers the typical case of a back-end application centered around a single schema containing all necessary tables, indices, etc. Most importantly, the business logic is implemented as stored code in the database.

The JavaScript implementation in the form of MLE modules and an MLE environment is encapsulated in a single schema. Access to the functionality is only exposed using MLE call specifications based on one or multiple modules. Users of the application are granted execute privileges on (PL/SQL) call specifications only. No further privileges on MLE modules and environment are granted, nor are they necessary.

Consequently, the owner of the MLE modules controls access to the application through the `AUTHID` clause attached to the MLE call specifications. The pseudo-code in Example 10-6 demonstrates this scenario.

Example 10-6 Business Logic Stored in MLE Modules

In this example, the application schema is referred to as `APP_OWNER`. Note how MLE modules and environments are restricted to the `APP_OWNER` schema.

```
-- MLE Module containing helper functions commonly used by the application
CREATE MLE MODULE app_owner.helper_module LANGUAGE JAVASCRIPT AS

export function setDebugLevel(level) {
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- An MLE Environment allowing other MLE Modules to import the helper module
CREATE MLE ENV app_owner.helper_module_env IMPORTS (
  'helperModule' module helper_module
);

-- The main application module imports the helper module for common tasks
CREATE MLE MODULE app_owner.orders_module LANGUAGE JAVASCRIPT AS

import { setDebugLevel } from "helperModule";

export function newOrder() {

setDebugLevel("INFO");
  // ... JavaScript code ...
}

export function delivery() {
  setDebugLevel("WARN");
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- The call specification is all the end users need to be granted
-- access to. The execute privilege to this definer's rights procedure
-- (created and executed with the app_owner's database privileges)
-- is all that needs granting to the application role.

CREATE app_owner.package orders_pkg AS

  PROCEDURE new_order AUTHID DEFINER AS
```

```
    MLE MODULE orders_module
    ENV helper_module_env
    SIGNATURE 'newOrder()';

  PROCEDURE delivery AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV helper_module_env
    SIGNATURE 'delivery()';

END order_pkg;
/

GRANT EXECUTE ON app_owner.package orders_pkg TO app_role;
```

Generic Data Processing Libraries

In this scenario, generic JavaScript functionality is logically grouped inside a database schema. The JavaScript code is neither functionally nor logically tied to any existing database objects. In other words, the processing logic is stateless.

As there is no relation to any database schema objects such as tables or views, object grants are of no concern. The JavaScript code purely transforms functional arguments. Examples for such libraries include machine learning code, image manipulation like scaling, cropping, changes of resolution, etc. Other use cases include input validation or JSON processing.

The main purpose of the MLE modules deployed in such a fashion is to provide you with a common set of JavaScript tools that can be used in your own applications. Therefore, there aren't any pre-defined MLE call specifications provided. Instead, the schema containing these modules grants the execute privilege on MLE modules. It is up to the grantee to define MLE call specifications matching the use case. If necessary, MLE environments can be created alongside the MLE modules with respective grants to developers wishing to use the functionality created. Example 10-7 illustrates this scenario.

Example 10-7 Generic Data Processing Libraries

```
-- Common functionality potentially referenced by multiple applications
-- is grouped in a database schema. This particular MLE Module provides
-- input validation
CREATE MLE MODULE library_owner.input_validator_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'input_validator.js');
/

-- Another MLE module provides common machine learning functionality
CREATE MLE MODULE library_owner.commom_ml_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'commom_ml_lib.js');
/

-- Rather than a Call Specification as demonstrated in Example 10-6,
-- this time the MLE Modules themselves are exported for use
-- in a different schema: frontend_app
GRANT EXECUTE ON library_owner.input_validator_module TO frontend_app;
GRANT EXECUTE ON library_owner.commom_ml_module TO frontend_app;

-- frontend_app makes explicit use of a select few functions exported
-- by the MLE modules
CREATE PACKAGE input_validator_pkg AS
```

```
  FUNCTION checkEMail(p_email VARCHAR2) RETURN BOOLEAN AS
    MLE MODULE library_owner.input_validator_module
    SIGNATURE 'checkEmail(string)';

  FUNCTION checkZIPCode(p_zipcode VARCHAR2) RETURN BOOLEAN AS
    MLE MODULE library_owner.input_validator_module
    SIGNATURE 'checkZIPCode(string)';

  -- additional functionality ...
END;
/
```

The grouping of common, stateless JavaScript code is not limited to a single schema. Further separation by feature, functionality, or maintainer is possible as well.

Generic Libraries in Business Logic

This scenario utilizes business logic contained in a single schema and extends functionality using generic libraries.

This example extends the scenarios demonstrated by Example 10-6 and Example 10-7. It is conceivable that the domain-specific business logic might require extension by common functionality such as logging or debugging. The latter can be written generically so that other applications can include it as well. There are numerous advantages to that approach including, but not limited to a unified framework for auxiliary functions.

In Example 10-8, the business logic in the `APP_OWNER`'s schema, defined in Example 10-6, is extended with the previously introduced validation and machine learning functionality from Example 10-7.

There is no "best way" to work with MLE modules and environments in the database. It always depends on your particular use case. The included examples simply provide some background on how application logic can be grouped or separated, depending on a project's needs.

Example 10-8 Use Generic Libraries in Business Logic

```
-- Centrally managed JavaScript code library in the LIBRARY_OWNER schema
CREATE MLE MODULE library_owner.commom_ml_module
  LANGUAGE JAVASCRIPT USING BFILE(js_src_dir, 'commom_ml_lib.js');
/

-- The grant makes the module available to APP_OWNER
GRANT EXECUTE ON library_owner.commom_ml_module TO app_owner;

-- Business logic in schema APP_OWNER makes use of the common ML library
CREATE MLE MODULE app_owner.helper_module LANGUAGE JAVASCRIPT AS

export function setDebugLevel(level) {
  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- A generic MLE environment references both APP_OWNER's as well as
```

```
-- LIBRARY_OWNER's MLE modules
CREATE MLE ENV app_owner.all_dependencies_env imports (
  'helperModule' module helper_module
  'commonML'     module library_owner.commom_ml_module
);

-- The main application module imports the helper module for common tasks
-- as well as the common machine learning module provided by LIBRARY_OWNER
CREATE MLE MODULE app_owner.orders_module LANGUAGE JAVASCRIPT AS

import { setDebugLevel } from "helperModule";
import { churnRate }     from "commonML";

export function newOrder() {

  setDebugLevel("INFO");
  // ... JavaScript code ...
}

export function delivery() {
  setDebugLevel("WARN");
  // ... JavaScript code ...
}

export function estimateChurnRate() {

  // This function was imported from the common ML library
  // (an MLE module not stored in APP_OWNERs schema)
  const cr = churnRate();

  // ... JavaScript code ...
}

// ... additional functionality ...
/

-- the call specification is all the end-users need to be granted
-- access to. The execute privilege to this definer rights procedure
-- (created and executed with the app_owner's database privileges)
-- is all that needs granting to the application role.

CREATE app_owner.package orders_pkg AS

  PROCEDURE new_order AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
    SIGNATURE 'newOrder()';

  PROCEDURE delivery AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
    SIGNATURE 'delivery()';

  FUNCTION estimateChurnRate AUTHID DEFINER AS
    MLE MODULE orders_module
    ENV all_dependencies_env
```

```
    SIGNATURE 'estimateChurnRate()';

END order_pkg;
/
```

A
MLE Type Conversions

Supported conversions between JavaScript and PL/SQL, SQL, and JSON data types.

JavaScript target types include both native JavaScript types as well as SQL wrapper types. Supported SQL types are converted to the analogous JavaScript type by default where such a natural counterpart exists. If a conversion is attempted and there is no corresponding JavaScript type, conversion to a native JavaScript type is not supported and values are instead converted to the corresponding SQL wrapper type by default.

> **Note:**
>
> MLE does not provide functionality to prevent information loss that might occur between conversions from a customized database character representation to the built-in string representation of JavaScript (UTF-16).

> **See Also:**
>
> - Server-Side JavaScript API Documentation for information about using `mle-js-bindings` to change the default mappings when exchanging values between PL/SQL and JavaScript
> - Server-Side JavaScript API Documentation for information on how to use `mle-js-plsqltypes` to create SQL wrapper types, such as `OracleNumber`
> - Server-Side JavaScript API Documentation for information on using `mle-js-oracledb` to override the default conversions (as seen in Table A-1) when fetching column values from a `SELECT` statement

Date Conversions

JavaScript `Date` represents an instant (i.e., a single moment in time). Conversions can occur between the instant type `Date` and PL/SQL types `DATE` and `TIMESTAMP` that do not have time zone information. Conversions between instants on the JavaScript side and `DATE` and `TIMESTAMP` on the other side are handled as follows:

- When converting a `Date` to a `TIMESTAMP` or `DATE`, the instant is converted to a timezone-aware datatime value in the current session time zone. The local datatime portion of this value is stored in the target `DATE` or `TIMESTAMP` value.
- To convert a `TIMESTAMP` or `DATE` to a timezone-aware `Date`, the source datetime value is interpreted to be in the session time zone and is converted into an instant according to the session time zone.

Table A-1 Supported Mappings from SQL and PL/SQL Types to JavaScript Types

SQL Type	JavaScript Types (Bold Font Signifies Default)
NUMBER	**number** OracleNumber
BINARY_FLOAT	**number**
BINARY_DOUBLE	**number**
BINARY_INTEGER[1]	**number**
BOOLEAN	**boolean**
VARCHAR2	**string**
NVARCHAR2	**string**
CHAR	**string**
NCHAR	**string**
CLOB	**OracleCLOB** string
NCLOB	**OracleCLOB** string
BLOB	**OracleBLOB** Uint8Array (TypedArray)
RAW	**Uint8Array** (TypedArray)
DATE	**Date** OracleDate
TIMESTAMP	**Date** OracleTimestamp
TIMESTAMP WITH TIME ZONE	**Date** OracleTimestampTZ
TIMESTAMP WITH LOCAL TIME ZONE	**Date** OracleTimestampTZ
INTERVAL YEAR TO MONTH	OracleIntervalYearToMonth
INTERVAL DAY TO SECOND	OracleIntervalDayToSecond
NULL[2]	**null**
JSON	any (object, array, null)[3]

1 Note that BINARY_INTEGER is a PL/SQL type and not supported in SQL. MLE only supports BINARY_INTEGER on PL/SQL interfaces.

2 Although not technically a type, MLE converts a SQL NULL value into a JavaScript null value and vice versa. This is so that JavaScript can indicate to the database that a value passed into the database is absent (for example, the return value of a function or an IN bind in a SQL statement).

3 See MLE JavaScript Support for JSON for details

Table A-2 Supported Mappings from JavaScript Types to SQL Types

JavaScript Type	SQL Type
number boolean OracleNumber	NUMBER
number	BINARY_FLOAT
number	BINARY_DOUBLE
number boolean	BINARY_INTEGER
number OracleNumber boolean	BOOLEAN
string	VARCHAR2
string	CHAR
string	NCHAR
string	NVARCHAR2
string OracleCLOB	CLOB
string OracleCLOB	NCLOB
string	UROWID
Uint8Array OracleBlob	BLOB
UintArray	RAW
Date OracleDate	DATE
Date OracleTimestamp	TIMESTAMP
Date OracleTimestampTZ	TIMESTAMP WITH (LOCAL) TIME ZONE
OracleIntervalYearToMonth	INTERVAL YEAR TO MONTH
OracleIntervalDayToSecond	INTERVAL DAY TO SECOND
null	NULL (any supported SQL type)

Table A-2 (Cont.) Supported Mappings from JavaScript Types to SQL Types

JavaScript Type	SQL Type
`number`	JSON[2]
`string`	
`boolean`	
`null`	
`undefined`	
`Date`	
`Uint8Array`	
`OracleNumber`	
`OracleDate`	
`OracleTimestamp`	
`OracleTimestampTZ`	
`OracleIntervalYearToMonth`	
`OracleIntervalDayToSecond`	
object[1]	

1 JavaScript objects and arrays that do not match one of the classes listed above

2 See MLE JavaScript Support for JSON for details

- MLE JavaScript Support for JSON
  Supported conversions between JavaScript and the JSON data type.
- MLE JavaScript Support for the VECTOR Data Type
  Oracle Multilingual Engine (MLE) supports conversions between JavaScript TypedArrays and SQL vectors with formats `INT8`, `FLOAT32`, and `FLOAT64`. Data exchanges between JavaScript and the `VECTOR` data type are supported by the MLE JavaScript SQL driver, MLE call specifications, and MLE JavaScript bindings.

MLE JavaScript Support for JSON

Supported conversions between JavaScript and the JSON data type.

Values of the SQL `JSON` type can be converted to and from JavaScript values. The type mapping between the SQL `JSON` type and JavaScript values is aligned with type mappings employed by the `node-oracledb` driver.

> **Note:**
>
> For more information about `node-oracledb` and the `JSON` data type, see the node-oracledb documentation.

Values of the SQL `JSON` type are converted to JavaScript values as follows:

- If the `JSON` value is an object, it is converted to an equivalent JavaScript object by converting all fields of the input object.

- If the `JSON` value is an array, it is converted to an equivalent JavaScript array by converting all elements of the input array.
- If the `JSON` value is a scalar value, it is converted to an equivalent value according to the type mapping in Table A-3.

Table A-3 Mapping from JSON Attribute Types and Values to JavaScript Types and Values

JSON Attribute Type or Value	JavaScript Type or Value
`null`	`null`
`false`	`false`
`true`	`true`
`NUMBER`	`Number`
`VARCHAR2`	`String`
`RAW`	`Uint8Array`
`CLOB`	`String`
`BLOB`	`UintArray`
`DATE`	`Date`
`TIMESTAMP`	`Date`
`INVERVAL YEAR TO MONTH`	`OracleIntervalYearToMonth`
`INTERVAL DAY TO SECOND`	`OracleIntervalDayToSecond`
`BINARY_DOUBLE`	`Number`
`BINARY_FLOAT`	`Number`
Arrays	Array
Objects	A plain JavaScript Object

Values of a JavaScript type are converted to the SQL `JSON` type as follows:

- If the JavaScript value matches one of the scalar types in the first column of Table A-4, it is converted to a JSON value of the corresponding type.
- If the JavaScript value is an array, it is converted to a JSON array by converting all elements of the array. Note that `Uint8Array` values are treated as scalars as opposed to arrays, so `Uint8Array` values are converted to the type `RAW`, not to a JSON array.
- If the JavaScript value is an object that is neither an array nor matches any of the JavaScript types/ classes listed in Table A-4, it is converted to a JSON object. Each field of the object is converted according to the appropriate mappings.

Table A-4 Mapping from JavaScript Types and Values to JSON Attributes and Values

JavaScript Type or Value	JSON Attribute Type or Value
`null`	`null`
`undefined`	`null`
`string`	`VARCHAR2`
`true`	`true`
`false`	`false`
`Uint8Array`	`RAW`
`Number`	`NUMBER`
`Date`	`DATE`

Table A-4 (Cont.) Mapping from JavaScript Types and Values to JSON Attributes and Values

JavaScript Type or Value	JSON Attribute Type or Value
`OracleNumber`	`NUMBER`
`OracleDate`	`DATE`
`OracleTimestamp`	`TIMESTAMP`
`OracleTimestampTZ`	`TIMESTAMP WITH TIME ZONE`
`OracleIntervalYearToMonth`	`INVERVAL YEAR TO MONTH`
`OracleIntervalDayToSecond`	`INTERVAL DAY TO SECOND`
Array	Array
Object	Object

MLE JavaScript Support for the VECTOR Data Type

Oracle Multilingual Engine (MLE) supports conversions between JavaScript TypedArrays and SQL vectors with formats `INT8`, `FLOAT32`, and `FLOAT64`. Data exchanges between JavaScript and the `VECTOR` data type are supported by the MLE JavaScript SQL driver, MLE call specifications, and MLE JavaScript bindings.

The `VECTOR` data type can appear as an `IN`, `OUT`, and `IN OUT` bind argument, as well as a return type. The `SIGNATURE` clause of an MLE call specification supports the following JavaScript types:

- `Float32Array`
- `Float64Array`
- `Int8Array`

Table A-5 Mapping from VECTOR Data Type to JavaScript Types

SQL Type	JavaScript Type
`VECTOR(*, float32)`	`Float32Array` (TypedArray)
`VECTOR(*, float64)`	`Float64Array` (TypedArray)
`VECTOR(*, int8)`	`Int8Array` (TypedArray)
`VECTOR(*)`	`Float64Array`[1] (TypedArray)

[1] When no vector format is specified, `Float64Array` is used by default

Table A-6 Mapping from JavaScript Types to VECTOR Data Type

JavaScript Type	SQL Type
`Float32Array`	`VECTOR(*, float32)`
`Float64Array`	`VECTOR(*, float64)`
`Int8Array`	`VECTOR(*, int8)`
`Array`	`VECTOR(*, float64)`

> **See Also:**
>
> - *Oracle Database AI Vector Search User's Guide* for more information about the `VECTOR` data type and Oracle AI Vector Search capabilities

Example A-1 Use VECTOR Data Type with MLE

This example demonstrates support of the `VECTOR` data type used in arguments and as return type in MLE call specifications.

```
SET SERVEROUTPUT ON;
CREATE OR REPLACE MLE MODULE vec_mod
LANGUAGE JAVASCRIPT AS

/**
 * Add two vectors
 * @param v1 the first vector
 * @param v2 the second vector
 * @returns the resulting vector after adding v1 and v2
 */
export function addVectors(v1, v2){
  return v1.map((element, index) => element + v2[index]);
}

/**
 * Subtract two vectors
 * @param v1 the first vector
 * @param v2 the second vector
 * @returns the resulting vector after subtracting v2 from v1
 */
export function subtractVectors(v1, v2){
  return v1.map((element, index) => element - v2[index]);
}
/

CREATE OR REPLACE PACKAGE mle_vec_pkg AS

  FUNCTION addVectors(
    input_vector1 IN VECTOR,
    input_vector2 IN VECTOR
  )
  RETURN VECTOR
    AS MLE MODULE vec_mod
    SIGNATURE 'addVectors';

  FUNCTION subtractVectors(
    input_vector1 IN VECTOR,
    input_vector2 IN VECTOR
  )
  RETURN VECTOR
    AS MLE MODULE vec_mod
    SIGNATURE 'subtractVectors';
```

```
END mle_vec_pkg;
/


SELECT mle_vec_pkg.addVectors(
  VECTOR('[1, 2]'),
  VECTOR('[3, 4]')
) AS result;
```

Result:

```
RESULT
--------------------------------------------
[4.0E+000,6.0E+000]


SELECT mle_vec_pkg.subtractVectors(
  VECTOR('[3, 4]'),
  VECTOR('[1, 2]')
) AS result;
```

Result:

```
RESULT
--------------------------------------------
[2.0E+000,2.0E+000]
```

Index

I

J

L

M

N

O

P

R

S

T

U

W