7

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


- Introduction to the MLE JavaScript SQL Driver
- Selecting Data Using the MLE JavaScript Driver
- Data Modification
- Bind Variables
- PL/SQL Invocation from the MLE JavaScript SQL Driver
- Error Handling in SQL Statements
- 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.
- Using Large Objects (LOB) with MLE
- 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.
- 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.


## 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 Fetch: Objects
- Fetching Rows as ResultSets: Arrays
- Fetching Rows as ResultSets: Iterating Over ResultSet Objects


# 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.
- RETURNING INTO Clause
- Batch Operations

## 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
- Positional Bind Variables

### 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.