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.
- OUT and IN OUT Parameters

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.
- Creating an Inline MLE Call Specification
  Inline MLE call specifications embed JavaScript code directly in the `CREATE FUNCTION` and `CREATE PROCEDURE` DDLs.
- Choosing Inline Versus Module MLE Call Specifications
  Each option provides its own advantages and disadvantages depending on your use case.
- Runtime Isolation for an MLE Call Specification
- 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.

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

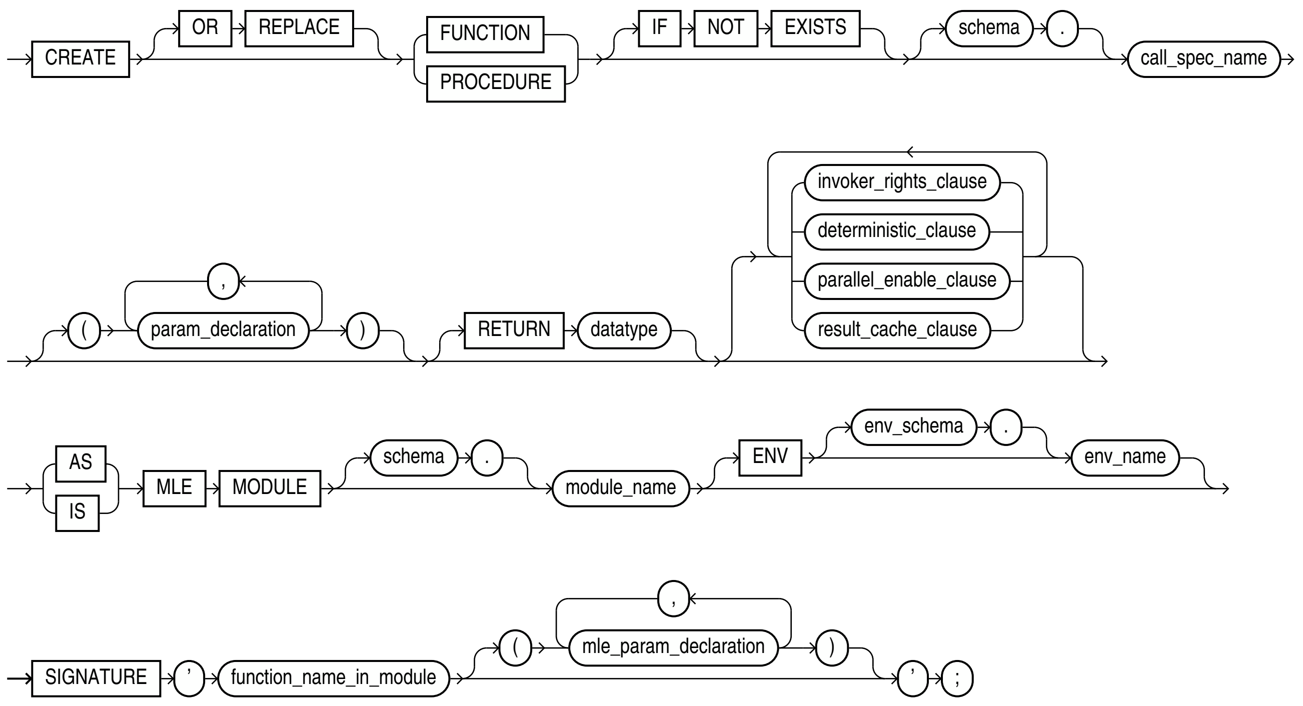


Table 6-1 Components of an MLE Call Specification

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 ::=

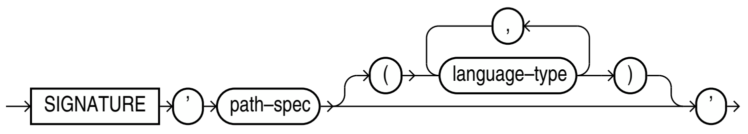


Figure 6-3 path_spec ::=

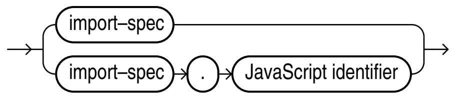
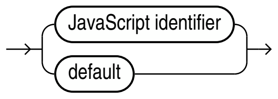


Figure 6-4 import_spec ::=



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

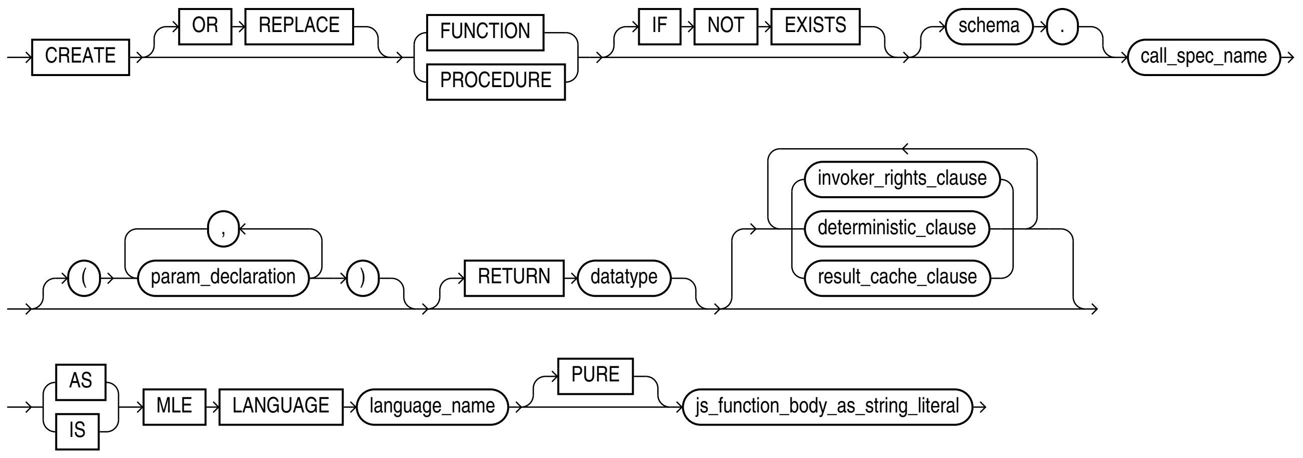


Table 6-2 Components of an Inline MLE Call Specification

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
```