3

MLE JavaScript Modules and Environments

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

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

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

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

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

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

Topics

Using JavaScript Modules in MLE

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

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

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

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

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

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

Topics

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

Managing JavaScript Modules in the Database

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

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

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

Topics

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

Naming JavaScript Modules

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

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

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

Creating JavaScript Modules in the Database

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

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

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

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

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

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

Example 3-1 Creating a JavaScript Module in the Database

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

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

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

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

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

Storing JavaScript Code in Databases Using Single-Byte Character Sets

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

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

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

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

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

Code Analysis

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

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

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

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

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

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

Preparing JavaScript code for MLE Module Calls

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

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

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

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

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

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

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

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

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

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

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

```
FROM
    j_purchaseorder po;
```

Result:

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

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

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

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

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

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

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

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

export { orderValue }
/
```

Additional Options for Providing JavaScript Code to MLE

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

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

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

Example 3-4 Providing JavaScript Source Code Using a BFILE

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

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

Example 3-5 Providing JavaScript Source Code Using a CLOB

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

As an alternative, you also have the option of using JavaScript source code that is stored in a table. This example variation assumes your schema features a table named `javascript_src` containing the JavaScript source code in column `src` along with some additional metadata. The following statement fetches the CLOB and creates the module.

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

```
)
/
```

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

Specifying Module Version Information and Providing JSON Metadata

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

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

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

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

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

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

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

Example 3-6 Specification of a VERSION string in CREATE MLE MODULE

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

Example 3-7 Addition of JSON Metadata to the MLE Module

This example uses the module `version_mod`, created in Example 3-6.

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

Drop JavaScript Modules

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

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

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

Example 3-8 Drop an MLE Module

```
DROP MLE MODULE unused_mod;
```

Example 3-9 Drop an MLE Module Using IF EXISTS

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

Alter JavaScript Modules

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

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

Example 3-10 Alter an MLE Module

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

Overview of Built-in JavaScript Modules

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

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

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

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

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

Dictionary Views Related to MLE JavaScript Modules

The Data Dictionary includes details about JavaScript modules.

Topics

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

USER_SOURCE

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

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

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

Example 3-11 Externalize JavaScript Module Source Code

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

Example output:

```
 LINE TEXT
----- ------------------------------------------------------------
    1 /**
    2  * calculate the value of a given line item. Factored out of the public
    3  * function to allow for currency conversions in a later step
    4  * @param {number} unitPrice - the price of a single article
    5  * @param {number} quantity - the quantity of articles ordered
    6  * @returns {number} the monetary value of the line item
    7  */
    8 function lineItemValue(unitPrice, quantity) {
    9     return unitPrice * quantity;
   10 }
   11
   12
   13 /**
   14  * get the value of all line items in an order
   15  * @param {array} lineItems - all the line items in a purchase order
   16  * @returns {number} the total value of all line items in a purchase
order
   17  */
   18 export function orderValue(lineItems) {
   19
   20     return lineItems
   21                 .map( x => lineItemValue(x.Part.UnitPrice, x.Quantity) )
   22                 .reduce(
   23                     (accumulator, currentValue) => accumulator +
currentValue, 0
   24                 );
   25 }
```

USER_MLE_MODULES

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

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

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

Example 3-12 Find MLE Modules Defined in a Schema

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

Example output:

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

Specifying Environments for MLE Modules

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

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

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

Topics

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

Creating MLE Environments in the Database

The SQL DDL supports the creation of MLE environments.

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

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

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

Topics

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

Naming MLE Environments

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

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

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

Creating an Empty MLE Environment

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

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

```
CREATE MLE ENV myEnv;
```

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

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

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

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

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

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

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

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

Creating an Environment as a Clone of an Existing Environment

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

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

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

Using MLE Environments for Import Resolution

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

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

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

Modules attempting to import functionality from other modules stored in the database require MLE environments in order to perform name resolution. To create an MLE environment with that information, the `IMPORTS` clause must be used. Example 3-13 demonstrates how a mapping is created between the identifier `po_module` and JavaScript module `PO_MODULE`, created in Example 3-1.

Multiple imports can be provided as a comma-separated list. In Example 3-13, the first parameter in single quotation marks is known as the import name. The import name is used by another module's import statement. In this case, `'po_module'` is the import name and refers to the module of the same name.

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

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

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

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

Example 3-13 Map Identifier to JavaScript Module

```
CREATE OR REPLACE MLE ENV
```

```
    po_env
IMPORTS (
    'po_module' MODULE PO_MODULE
);
```

Example 3-14 Import Module Functionality

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

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

    const purchaseOrder = {
        "PONumber": 1600,
        "Reference": "ABULL-20140421",
        "Requestor": "Alexis Bull",
        "User": "ABULL",
        "CostCenter": "A50",
        "ShippingInstructions": {
            "name": "Alexis Bull",
            "Address": {
                "street": "200 Sporting Green",
                "city": "South San Francisco",
                "state": "CA",
                "zipCode": 99236,
                "country": "United States of America"
            },
            "Phone": [
                {
                    "type": "Office",
                    "number": "909-555-7307"
                },
                {
                    "type": "Mobile",
                    "number": "415-555-1234"
                }
            ]
        },
        "Special Instructions": null,
        "AllowPartialShipment": true,
        "LineItems": [
            {
                "ItemNumber": 1,
                "Part": {
                    "Description": "One Magic Christmas",
                    "UnitPrice": 19.95,
```

```
                    "UPCCode": 13131092899
                },
                "Quantity": 9.0
            },
            {
                "ItemNumber": 2,
                "Part": {
                    "Description": "Lethal Weapon",
                    "UnitPrice": 19.95,
                    "UPCCode": 85391628927
                },
                "Quantity": 5.0
            }
        ]
    };

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


purchaseOrderValue


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

SELECT purchase_order_value;
/
```

Result:

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

Providing Language Options

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

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

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

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

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

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

Topics

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

JavaScript Language Options

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

Table 3-1 JavaScript Language Options

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

Dropping MLE Environments

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

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

```
DROP MLE ENV myOldEnv;
```

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

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

Modifying MLE Environments

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

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

Topics

- Altering Language Options
  You can modify language options provided to an MLE module.

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

Altering Language Options

You can modify language options provided to an MLE module.

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

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

Modifying Module Imports

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

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

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

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

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

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

Dictionary Views Related to MLE JavaScript Environments

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

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

Topics

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

USER_MLE_ENVS

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

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

Example 3-15 List Available MLE Environments Using USER_MLE_ENVS

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

Example SQL*Plus output:

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

USER_MLE_ENV_IMPORTS

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

MLE environments are the key enablers for resolving names of imported modules. Example 3-16 demonstrates a query against `USER_MLE_ENV_IMPORTS` to list `IMPORT_NAME`, `MODULE_OWNER`, and `MODULE_NAME`.

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

Example 3-16 List Module Import Information Using USER_MLE_ENV_IMPORTS

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

SQL*Plus output:

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