5

Overview of Importing MLE JavaScript Modules

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

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

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

There are two options available for module imports in MLE:

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

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

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

Topics

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

JavaScript Module Hierarchies

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

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

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

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

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

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

Topics

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

Resolving Import Names Using MLE Environments

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

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

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

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

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

CREATE OR REPLACE MLE MODULE mod_object_import_mod LANGUAGE JAVASCRIPT AS

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

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

Export Functionality

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

Topics

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

Named Exports

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

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

Example 5-2 Function Export using Named Exports

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

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

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

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

export {sum, difference};
/
```

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

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

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

Example 5-3 Function Export Using Export Keyword Inline

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

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

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

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

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

Default Exports

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

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

Example 5-4 Export a Class Using a Default Export

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

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

export default class myMath {

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

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

Private Identifiers

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

Example 5-5 Named Export of Single Function

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

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

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

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

export { sum };
/
```

Import Functionality

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

Topics

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

Module Objects

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

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

Example 5-6 Module Object Definition

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

CREATE OR REPLACE MLE MODULE mod_object_import_mod
LANGUAGE JAVASCRIPT AS

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

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

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

export {mySum, myDifference};
/
```

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

Named Imports

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

Example 5-7 Named Imports Using Specified Identifiers

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

import {sum, difference} from "namedExports";

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

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

export {mySum, myDifference};
/
```

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

Example 5-8 Named Imports with Aliases

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

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

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

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

export {mySum, myDifference};
/
```

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

Default Imports

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

Example 5-9 Default Import

This example demonstrates the default import of `myMathClass`.

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

CREATE MLE MODULE default_import_module LANGUAGE JAVASCRIPT AS

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

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

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

Example 5-10 Default Import with Built-in Module

```
CREATE MLE MODULE default_import_built_in_mod
LANGUAGE JAVASCRIPT AS

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

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

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

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

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

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